

计算机科学导论

跨学科方法

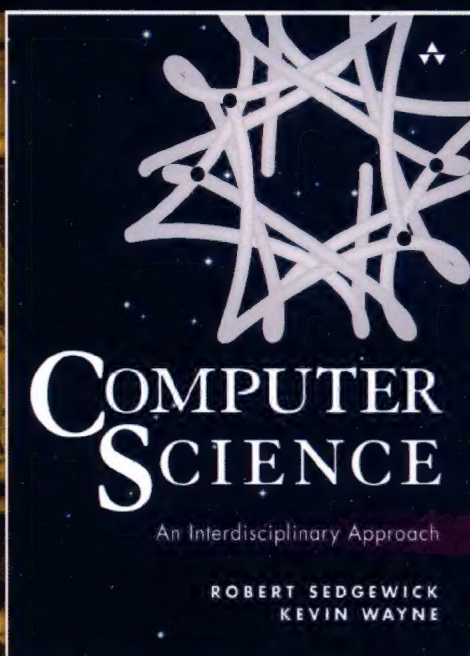
[美] 罗伯特·塞奇威克 (Robert Sedgewick) 凯文·韦恩 (Kevin Wayne) 著

普林斯顿大学

宫晓利 郭宇飞 曹丁元 张金 译

南开大学

普林斯顿大学历经近30年打造的计算机科学入门课



Computer Science
An Interdisciplinary Approach



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

计算机科学导论

跨学科方法

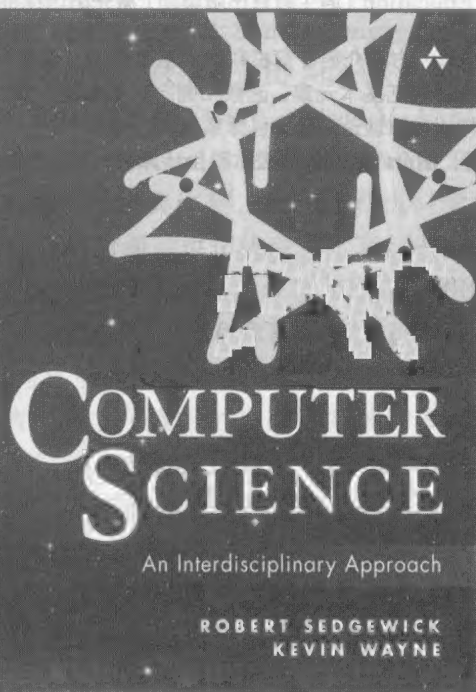
[美] 罗伯特·塞奇威克 (Robert Sedgewick) 凯文·韦恩 (Kevin Wayne) 著

普林斯顿大学

宫晓利 郭宇飞 曹丁元 张金 译

南开大学

Computer Science
An Interdisciplinary Approach



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

计算机科学导论: 跨学科方法 / (美) 罗伯特·塞奇威克 (Robert Sedgewick), (美) 凯文·韦恩 (Kevin Wayne) 著; 宫晓利等译. —北京: 机械工业出版社, 2020.1
(计算机科学丛书)

书名原文: Computer Science: An Interdisciplinary Approach

ISBN 978-7-111-64141-4

I. 计… II. ① 罗… ② 凯… ③ 宫… III. 计算机科学 IV. TP3

中国版本图书馆 CIP 数据核字 (2019) 第 252856 号

本书版权登记号: 图字 01-2016-8658

Authorized translation from the English language edition, entitled Computer Science: An Interdisciplinary Approach, ISBN: 978-0-13-407642-3, by Robert Sedgewick, Kevin Wayne, published by Pearson Education, Inc., Copyright © 2017 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by China Machine Press, Copyright © 2020.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括香港、澳门特别行政区及台湾地区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

本书面向初学者, 以跨学科的方法介绍计算机科学的基本知识。全书包括两个部分, 第一部分围绕学习编程的三个阶段进行组织, 包括基本元素、函数和面向对象编程。第二部分则介绍计算机科学的高级主题, 包括算法和数据结构、计算理论和计算机体系结构。

本书内容丰富、循序渐进, 适合作为高等院校本科生计算机科学入门的教材。

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 余 洁

责任校对: 殷 虹

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2020 年 1 月第 1 版第 1 次印刷

开 本: 185mm×260mm 1/16

印 张: 44.25

书 号: ISBN 978-7-111-64141-4

定 价: 139.00 元

客服电话: (010) 88361066 88379833 68326294

投稿热线: (010) 88379604

华章网站: www.hzbook.com

读者信箱: hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来,源远流长的科学精神和逐步形成的学术规范,使西方国家在自然科学的各个领域取得了垄断性的优势;也正是这样的优势,使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中,美国的产业界与教育界越来越紧密地结合,计算机学科中的许多泰山北斗同时身处科研和教学的最前线,由此而产生的经典科学著作,不仅擘划了研究的范畴,还揭示了学术的源变,既遵循学术规范,又自有学者个性,其价值并不会因年月的流逝而减退。

近年,在全球信息化大潮的推动下,我国的计算机产业发展迅猛,对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇,也是挑战;而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下,美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此,引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用,也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始,我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力,我们与Pearson、McGraw-Hill、Elsevier、MIT、John Wiley & Sons、Cengage等世界著名出版公司建立了良好的合作关系,从它们现有的数百种教材中甄选出Andrew S. Tanenbaum、Bjarne Stroustrup、Brian W. Kernighan、Dennis Ritchie、Jim Gray、Afred V. Aho、John E. Hopcroft、Jeffrey D. Ullman、Abraham Silberschatz、William Stallings、Donald E. Knuth、John L. Hennessy、Larry L. Peterson等大师名家的一批经典作品,以“计算机科学丛书”为总称出版,供读者学习、研究及珍藏。大理石纹理的封面,也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助,国内的专家不仅提供了中肯的选题指导,还不辞劳苦地担任了翻译和审校的工作;而原书的作者也相当关注其作品在中国的传播,有的还专门为其书的中译本作序。迄今,“计算机科学丛书”已经出版了近500个品种,这些书籍在读者中树立了良好的口碑,并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化,教育界对国外计算机教材的需求和应用都将步入一个新的阶段,我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方式如下:

华章网站: www.hzbook.com

电子邮件: hzsj@hzbook.com

联系电话: (010) 88379604

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037



华章教育

华章科技图书出版中心

信息革命带来了科技的突飞猛进,使我们前所未有地感受到了技术的力量。究其本源,当前的各种便利无不依赖于电子计算体系所支撑的强大算力。编程也好,算法也罢,无外乎对于算力的驾驭问题。

细说起来,编程又分为编程语言和编程思想。如果把编程语言比作文武招式,那么编程思想就是内力。按照传统计算机知识体系,以熟悉一门编程语言作为起手式和基本功,继而开始学习数据结构、各种算法设计、计算机组成原理、编译原理等知识模块,再深入研究可计算性原理、形式语言与自动机等内容。如此从运用入手继而回溯计算机构造方法乃至思想的路径,虽然有助于循序渐进地了解 and 掌握程序的运行规律、编程语言的设计思想,以便深入理解如何写出优秀的代码。但是,这一路径的每个知识模块背后都是数部艰深的巨著,对于那些希望短时间内能够掌握程序设计思想和语言运用的学习者来说,这趟计算机寻源之旅显得有些漫长。

而本书恰为上述读者提供了一条“捷径”,作者是来自普林斯顿大学的两位大师 Robert Sedgewick 和 Kevin Wayne,书中巧妙地区分了“程序设计思想”与“编程语言使用”之间的微妙关系,通过对 Java 语言的学习,讲述了通过程序解决问题的重要思路,而没有拘泥于 Java 的语法和语言特性。更为精彩的是,本书在后半部分构建了一台“玩具型”计算机,并分析了它如何执行前半部分写出的程序。最终把我们熟知的各类知识和应用场景,与计算机深层的工作原理和设计方法结合在了一起。本书已经被普林斯顿大学等多所国际知名大学广泛采用,堪称好评如潮。此次我们配合机械工业出版社引进、翻译本书,就是希望能够为想要攀登计算机这座新世纪高峰的读者提供一条新的便捷之道。

本书能够顺利成稿,要衷心感谢南开大学嵌入式系统与信息安全实验室的孙承君、刘希明、刘振、尹腾召、赵洋、闫美君、权玮虹、曹丁元、李浩然等同学在全书翻译过程中付出的辛苦努力;感谢南开大学计算机学院与网络空间安全学院的各位老师在本书翻译过程中提供的指导和支持;要感谢机械工业出版社的各位同仁的鼎力协助。本书的若干个相似版本均有前辈或同仁进行了翻译,阅读这些译著的过程令我们受益匪浅,特此表示感谢。限于译者水平和经验,译文中难免存在不当之处,恳请读者提出宝贵意见。

最后,还要感谢微微和点点两位小朋友,要不是你们,生活不会如此快乐;要不是你们,这本书估计早就翻译完了吧!

译者于马蹄湖畔

2019年9月

20 世纪的教育基础是“阅读、写作和算术”，现在则是“阅读、写作和计算”。学习编程是科学和工程领域教育的重要组成部分。除了直接应用外，这是理解计算机科学的第一步，进而能够理解为什么计算机会对现代世界产生如此巨大的影响。本书的目的是在科学的应用环境中讲解编程的相关知识。

我们的主要目标是通过提供有效使用计算所需的经验和基本工具来增强学生的能力。我们的方法是教会学生按照一种自然的、令人满意的、创造性的方式编写程序。我们逐步引入基本概念，并引入应用数学和科学领域的经典应用来说明概念，同时为学生提供编写程序来解决问题的机会。我们也设法帮助学生揭开计算的神秘面纱，使他们建立对计算机科学领域的重要知识的基本认识。

本书中的所有程序都使用 Java 编程语言编写。本书的第一部分教授解决计算问题的基本技能，所使用的编程方法适用于许多现代计算环境，这是一个完整的解决方案，即使没有编程经验的人也能够学会。这里我们强调的是关于编程的基本概念，而不是 Java 本身。本书的第二部分更多地偏重计算机科学的知识而不再是编程，但是我们仍然经常使用 Java 程序来交流主要想法。

本书是按照跨学科的方法对传统 CS1 (computer science) 课程进行扩充，我们强调计算在其他学科中的作用，从材料科学到基因组学、天体物理学和网络系统。这种方法会强化学生对于“数学、科学、工程和计算在现代世界中交织在一起”的基本认识。虽然本书是为一年级大学生设计的 CS1 教科书，但也可用于自学。

范围 本书的第一部分围绕学习编程的三个阶段进行组织：基本元素、函数、面向对象编程。我们提供了读者在进入下一个层次之前需要熟练掌握的基本内容。我们的方法的一个基本特征是使用示例程序来解决有趣问题，并辅以练习，包括从自学练习到需要创造性解决方案的挑战性问题。

基本元素包括变量、赋值语句、内置数据类型、控制流、数组和输入/输出（包括图形和声音等）。

函数是学生首次接触模块化编程的内容。我们假设学生已经熟悉数学函数，在此基础上引入 Java 函数，然后考虑函数编程的意义，包括函数库和递归。我们强调编程时要将程序划分为可独立调试、可维护和可复用的组件，这是编程的基本思路。

面向对象编程是关于对数据抽象的介绍。我们强调数据类型的概念及其使用 Java 类机制的实现。我们教授学生如何使用、创建和设计数据类型。模块化、封装和其他现代编程范例是这个阶段的中心概念。

本书的第二部分介绍计算机科学的高级主题：算法和数据结构、计算理论和计算机体系结构。

算法和数据结构将现代编程范例与组织和处理数据的经典方法相结合，这些经典方法在现代应用中仍然有效。我们介绍了用于排序和搜索的经典算法、基本数据结构及其应用，并且强调了如何使用科学方法来理解某段代码的性能特征。

计算理论使用简单的计算机抽象模型，帮助我们解决计算的基本问题。这些知识不仅具有重要的理论意义，而且许多想法在实际的计算应用中也是非常相关甚至可以直接应用的。

计算机体系结构提供了一个理解真实世界中实际计算的途径——在计算理论的抽象计算设备和我们使用的真实计算机之间建立了联系。而且，体系结构的研究还提供了与过去的联系，因为当今计算机和移动设备中的微处理器与 20 世纪中叶开发的第一台计算机中的没有太大区别。

本书的关键特征之一是注重编程在科学和工程中的应用。我们通过分析编程对特定应用领域产生的巨大影响来引导和激发学生学习每个相应的编程概念，以应用数学、物理和生物科学以及计算机科学本身为例，涉及的问题包括物理系统模拟、数值方法、数据可视化、声音合成、图像处理、财务模拟和信息技术等。具体的例子包括第 1 章中基于马尔可夫链的网页排名处理，以及后续章节用于解决渗透问题、多体模拟和小世界现象的案例分析。这些应用程序是本书的重要组成部分。它们能够引导学生学习相关主题，说明编程概念的重要性，并为计算在现代科学与工程领域所发挥的重要作用提供有说服力的证据。

在后面的章节中强调了编程发展的历史知识，也着重介绍了艾伦·图灵、冯·诺依曼等人关于计算的基本思想发展和应用的有趣故事。

我们的主要目标是让学生掌握有效解决任何编程问题所需的特定机制和技能。书中使用的代码都是完整的 Java 程序，读者可以试着使用它们来学习编程。需要说明的是，本书专注于个人编程，并没有涉及大型编程问题。

如何使用本书 本书适合科学应用类相关专业一年级本科生学习计算机科学课程使用。使用本书时，大学生可以在相对熟悉的专业背景下学习编程。通过本书和相关课程的学习，学生能够熟练地将编程技能应用到他们所选专业的后续课程中，并能够认识到深入学习计算机科学是很有意义的。

对于计算机科学专业的学生，在科学应用的背景下学习编程也会受益良多。为了更好地从事科研工作，计算机科学家需要具备关于科学方法的基本知识，也需要了解计算在科学研究中的作用，并应该与生物学家、工程师或物理学家等对于这类问题的认识保持一致。

事实上，我们的跨学科方法可以让计算机科学专业的学生和其他专业的学生在同一门课程中学习编程。我们涵盖了 CS1 要求的所有内容，而且更注重通过具体应用引出编程的相关概念并吸引学生学习。同时，我们的跨学科方法能够让学生接触到来自不同学科的多种问题，从而帮助他们更明智地选择自己的专业。

无论使用哪种具体的机制，本书都适合在整个大学教学过程的初期使用。首先，这种教学时间的安排使得我们能够利用高中的数学和科学相关知识开展教学；其次，在进入专业时，在大学课程初期学习了编程的学生将能够更有效地使用计算机。就像阅读和写作一样，编程对于任何科学家或工程师来说肯定都是必不可少的技能。掌握了本书相关概念的学生将会在他们的一生中不断地发展这种技能，从而能够更好地理解在他们所选择的领域中出现的问题和项目，并能够更好地利用计算机来解决它们。

前导课程 本书适合一年级本科生。也就是说，只需要科学和数学的入门级基础知识，而不再要求其他额外的知识作为前导。

熟练掌握数学工具和知识是学习本书的重要前提。虽然我们不讨论数学的细节内容，但是确实涉及了学生高中所学的数学课程，包括代数学、几何学和三角学。我们认为本书的目标受众基本掌握了这些知识。事实上，我们的很多编程概念建立在他们相对熟悉的初等数学

课程的基础上。

科学好奇心也是一个重要的组成部分。理工科学生天然就迷恋科学探究，喜欢探究自然界发生的一切。我们利用他们的这种爱好，在书中通过简单的程序揭示自然界的规律。除了高中课程所讲授的数学、物理、生物或化学的基础知识以外，我们不需要任何特定的知识。

编程经验不是必需的，如果有编程经验也无妨。讲授编程是我们的主要目标之一，所以我们假设学生都没有编程经验。这是一本入门性质的编程书，但是那些在高中编写了大量程序的学生也可以在学习中有收获，因为本书中涉及的问题都是跨学科的新问题，而通过编程解决新问题往往是具有挑战性的智力任务。本书适合具有不同背景的学生阅读，因为这些应用程序对新手和专家都有吸引力。

学生不必具备使用计算机的经验。当然，大学生经常使用计算机。例如，与亲友交流、听音乐、处理照片，等等。在本书的学习中，他们将意识到能够以更有趣而且更加重要的方式利用自己的计算机，这会是一个激动人心同时又持久的过程。

总之，几乎所有的大学生都可以在第一学期的课程中学习本书的内容。

目标 对于那些完成了本书课程的学生，当他们开始学习理工科的高年级课程时，他们的教师会对他们有什么期待呢？

本书涵盖了 CS1 课程，但是任何教过编程基础课程的人都知道，在后续课程中，教师的期望通常很高：每位教师都希望所有的学生能够熟悉计算机环境和课程中所要使用的方法。物理学教授可能会期望学生在周末设计一个程序来进行模拟，工科教授可能会期望学生使用一个特定的包来实现微分方程的数值求解，计算机科学教授可能期望学生了解特定编程环境的细节。一个入门级课程满足如此多样化的期望，这现实吗？每类学生都应该有不同的入门课程吗？

自 20 世纪后半期计算机广泛使用以来，高等院校一直受这些问题的困扰。我们利用本书给出了这些问题的答案。这是一本介绍通用编程方法的教材，类似于普遍接受的数学、物理、生物和化学等学科的入门课程。本书致力于为所有理工科学生提供所需的基础知识，同时传递出一个清晰的信号，即对计算机科学的了解远不止程序设计这么简单。完成了本书的学习，我们相信学生将具备必要的知识和经验，能够适应新的计算环境，并在不同的应用程序中有效地利用计算机。

对于学生而言，学完了本书对应的课程之后，在后续的学习中可以尝试什么新的课程呢？

我们认为编程并不难学，而且学会利用计算机的力量是非常有价值的。掌握了本书内容的学生，以后将有能力应对职业生涯中出现的各类计算机使用上的挑战。他们学习了如 Java 提供的现代编程环境，有助于打开任何以后可能遇到的计算问题的大门，并有足够的信心学习、评估和使用其他新的计算工具。对计算机科学感兴趣的学生能够具备深入学习的基础知识，其他理工科学生则可具备将计算融入其他研究中的能力。

本书官网^① 在下面的网站中，可以找到本书的大量补充信息和其他相关材料：

<http://introcs.cs.princeton.edu/java>

为了简单起见，我们以后不再写出网址，而简写为本书官网。网站上提供了面向教师、学生和普通读者的材料。我们在这里对这些材料进行简要介绍，读者可以进入网站浏览和查看详细内容。除了一些用于考试的材料外，其他材料都是公开的。

① 本书官网及网站上的资源由原书作者提供和维护，资源的可获取性、准确性、安全性由网站所有者负责，我社不承担任何责任。——编辑注

建立本书官网最重要的意义之一就是帮助教师和学生使用自己的计算机来讲授和学习本书中的材料。任何拥有计算机和浏览器的人都可以按照本书官网上列出的方法开始学习编程。这个过程不会比下载媒体播放器或歌曲更困难。与其他网站一样,本书官网也在不断发展。对于拥有本书的每个人来说,这是必不可少的资源。需要特别说明的是,网站上提供了更多的补充材料,这些材料对于学习本书知识至关重要,能够促进计算机科学成为所有科学家和工程师所接受的基础教育的一个重要组成部分。

对于教师来说,该网站包含有关教学的信息。在过去十年中,我们每周组织两次大规模的授课,并辅以每周两次的讨论,学生以小组为单位与教师或助教进行交流。在这些教学过程中,我们形成了自己的一套教学风格,并按照这套风格组织了教学材料。本书官网中给出了授课的演示幻灯片。

对于助教来说,这个网站包含了详细的习题集和编程任务,这些都是与书中的练习相对应的,而且加入了更多的细节。每个编程任务都会在一个有趣的应用环境中介绍一个相关的概念,同时向学生提出一个充满吸引力又有些难度的问题。任务难度的递进体现了我们的教学方法。本书官网详细说明了所有的作业,并提供了详细的、条理清晰的辅助材料,以帮助学生在规定时间内完成它们。辅助材料包括建议的解决方法,以及在讨论课上讲授的内容大纲。

对于学生来说,通过该网站可以快速访问本书中的大部分内容,包括源代码以及一些建议自学的课外材料。本书官网也提供了许多书中习题的答案,包括完整的程序代码和测试数据。网站上还有大量与编程任务相关的信息,包括建议的方法、清单、常见问题解答和测试数据等。

对于普通读者来说,可以通过本书官网访问与书籍内容相关的所有额外信息。所有网站上的内容都提供了链接和其他信息渠道,以供读者获取关于该主题的更多信息。网站上的内容非常丰富,远远超出了读者的需求,我们的目标是提供足够多的信息,确保能够满足每位读者对本书内容深入学习的需求。

致谢 这个项目自1992年启动以来一直处于不断的发展中,到目前为止,有太多的人为本书的成功做出了贡献,我们在这里感谢这一切。特别感谢 Anne Rogers 从开始一直帮助这个项目;感谢 Dave Hanson、Andrew Appel 和 Chris van Wyk 耐心地解释数据抽象;感谢 Lisa Worthington 和 Donna Gabai 最早尝试向一年级学生讲授本教材,这是一个非常大的挑战;感谢 Doug Clark 耐心帮助我们完善了图灵机构建和电路的知识。我们也非常感谢 /dev/126[⊖] 的努力;感谢普林斯顿大学25年来一直致力于讲授这本教材的教师、研究生和教学人员,以及成千上万名努力学习本书的大学生。

⊖ 作者所在研究小组的名称。——译者注

第1章 编程元素.....1

程序 1.1.1 Hello, World.....2

程序 1.1.2 使用命令行参数.....4

程序 1.2.1 字符串连接.....11

程序 1.2.2 整数的乘法与除法.....13

程序 1.2.3 求解一元二次方程.....15

程序 1.2.4 闰年.....17

程序 1.2.5 通过类型转换得到一个随机
整数.....21

程序 1.3.1 翻转硬币.....31

程序 1.3.2 第一个 while 循环语句.....32

程序 1.3.3 计算 2 的幂.....33

程序 1.3.4 你的第一个嵌套循环程序.....37

程序 1.3.5 谐波数 (一).....39

程序 1.3.6 牛顿法.....40

程序 1.3.7 二进制转换.....41

程序 1.3.8 赌徒破产模拟.....43

程序 1.3.9 整数分解.....44

程序 1.4.1 无须交换的采样.....60

程序 1.4.2 模拟卡券收集 (一).....63

程序 1.4.3 埃拉托斯特尼筛法.....64

程序 1.4.4 自回避的随机游走.....69

程序 1.5.1 生成一个随机序列.....77

程序 1.5.2 交互式用户输入.....82

程序 1.5.3 求数字流的均值.....83

程序 1.5.4 一个简单的过滤器.....84

程序 1.5.5 标准输入到绘图过滤器.....88

程序 1.5.6 弹跳球.....92

程序 1.5.7 数字信号处理.....95

程序 1.6.1 计算转移矩阵.....103

程序 1.6.2 模拟一个随机冲浪者.....105

程序 1.6.3 混合马尔可夫链.....109

第2章 函数和模块.....113

程序 2.1.1 谐波数 (二).....114

程序 2.1.2 高斯函数.....121

程序 2.1.3 模拟卡券收集 (二).....122

程序 2.1.4 按调演奏 (改进版).....126

程序 2.2.1 随机数库.....138

程序 2.2.2 数组 I/O 库.....141

程序 2.2.3 迭代函数系统.....143

程序 2.2.4 数据分析库.....145

程序 2.2.5 绘制数组中的数据值.....147

程序 2.2.6 伯努利试验.....148

程序 2.3.1 欧几里得算法.....158

程序 2.3.2 汉诺塔问题.....160

程序 2.3.3 格雷码.....163

程序 2.3.4 递归图形.....164

程序 2.3.5 布朗桥.....166

程序 2.3.6 最长公共子序列.....171

程序 2.4.1 渗透程序的脚手架.....179

程序 2.4.2 垂直渗透检测.....180

程序 2.4.3 可视化显示的客户端.....182

程序 2.4.4 渗透概率估计.....183

程序 2.4.5 渗透检测.....185

程序 2.4.6 自适应绘图客户程序.....186

第3章 面向对象编程.....193

程序 3.1.1 识别潜在基因.....198

程序 3.1.2 亚伯斯正方形.....201

程序 3.1.3 亮度模块.....203

程序 3.1.4 将彩色图像转为灰度图像.....205

程序 3.1.5 图像缩放.....207

程序 3.1.6 淡化效果.....208

程序 3.1.7 拼接文件.....210

程序 3.1.8 股票报价的 Web 信息抓取.....212

程序 3.1.9	分割一个文件	213	程序 4.4.1	字典查找	367
程序 3.2.1	带电粒子	227	程序 4.4.2	索引	368
程序 3.2.2	秒表	230	程序 4.4.3	散列表	371
程序 3.2.3	直方图	231	程序 4.4.4	二叉搜索树	377
程序 3.2.4	海龟绘图	233	程序 4.4.5	去重过滤器	381
程序 3.2.5	等角螺线	235	程序 4.5.1	图数据类型	393
程序 3.2.6	复数	239	程序 4.5.2	使用图来反转索引	396
程序 3.2.7	曼德布洛特集合	241	程序 4.5.3	最短路径客户程序	398
程序 3.2.8	股票账户	244	程序 4.5.4	最短路径实现	403
程序 3.3.1	复数 (另一种实现方式)	255	程序 4.5.5	小世界测试	406
程序 3.3.2	计数器	257	程序 4.5.6	演员 - 演员图	407
程序 3.3.3	空间向量	261			
程序 3.3.4	文档摘要	271	第 5 章 计算理论		415
程序 3.3.5	相似性检测	273	程序 5.1.1	有效性检查 (正则表达式识别)	424
程序 3.4.1	引力体	282	程序 5.1.2	广义 RE 模式匹配	428
程序 3.4.2	多体模拟	284	程序 5.1.3	通用虚拟 DFA	433
第 4 章 算法和数据结构		287	程序 5.2.1	虚拟图灵机的磁带	455
程序 4.1.1	三数求和问题	289	程序 5.2.2	通用虚拟 TM	456
程序 4.1.2	验证倍增假设	290	程序 5.5.1	SAT 求解器	501
程序 4.2.1	二分查找法 (20 个问题 游戏)	309	第 6 章 构建一台计算机		509
程序 4.2.2	二分查找法	311	程序 6.1.1	将一个自然数从一种进制 转换为另一种进制	515
程序 4.2.3	二分查找 (在一个排好序的 数组中)	312	程序 6.1.2	提取浮点数的组成部分	523
程序 4.2.4	插入排序	317	程序 6.2.1	你的第一个 TOY 程序	535
程序 4.2.5	插入排序算法的倍增测试	318	程序 6.2.2	条件和循环: 欧几里得 算法	539
程序 4.2.6	归并排序	320	程序 6.2.3	自修改代码: 计算一串 数字的和	540
程序 4.2.7	频率计数	323	程序 6.3.1	调用函数: 检测素数	546
程序 4.3.1	字符串 (数组) 栈	330	程序 6.3.2	标准输出: 斐波那契数列	548
程序 4.3.2	字符串 (链表) 栈	333	程序 6.3.3	标准输入: 求和计算	549
程序 4.3.3	字符串 (可变数组) 栈	336	程序 6.3.4	数组处理: 读取数组	550
程序 4.3.4	泛型栈	339	程序 6.3.5	链接结构: 在 BST 中 搜索 / 插入	552
程序 4.3.5	表达式求值	341	程序 6.4.1	TOY 虚拟机 (无标准输入 输出)	565
程序 4.3.6	通用 FIFO 队列 (链表)	345			
程序 4.3.7	M/M/1 队列模拟	348			
程序 4.3.8	负载均衡模拟	354			

第 7 章 构建计算设备

基本逻辑门

选择多路复用器

解码器

多路分配器

多路复用器

XOR

表决器

奇偶校验

加法器

ALU

总线多路复用器

SR 触发器

寄存器位

寄存器

内存位

内存

时钟

程序计数器

控制

中央处理器

目 录

Computer Science: An Interdisciplinary Approach

出版者的话

译者序

前言

程序列表

电路列表

第 1 章 编程元素.....1

1.1 你的第一个程序.....1

1.2 内置数据类型.....7

1.3 条件语句与循环语句.....29

1.4 数组.....55

1.5 输入 / 输出.....76

1.6 案例研究：随机网络冲浪.....101

第 2 章 函数和模块.....113

2.1 函数的定义.....113

2.2 库和客户程序.....133

2.3 递归.....154

2.4 案例研究：渗透.....176

第 3 章 面向对象编程.....193

3.1 使用数据类型.....193

3.2 创建数据类型.....224

3.3 设计数据类型.....252

3.4 案例研究：多体模拟.....279

第 4 章 算法和数据结构.....287

4.1 性能.....287

4.2 排序和搜索.....308

4.3 栈和队列.....327

4.4 符号表.....362

4.5 案例研究：小世界现象.....389

第 5 章 计算理论.....415

5.1 形式语言.....416

5.2 图灵机.....447

5.3 普遍性.....460

5.4 可计算性.....471

5.5 难解性.....480

第 6 章 构建一台计算机.....509

6.1 信息表示.....509

6.2 TOY 计算机.....529

6.3 机器语言编程.....544

6.4 TOY 虚拟机.....559

第 7 章 构建计算设备.....574

7.1 布尔逻辑.....574

7.2 基本电路模型.....583

7.3 组合电路.....589

7.4 时序电路.....610

7.5 数字设备.....623

后记.....637

术语表.....639

索引.....645

API.....688

编程元素

阅读完本章，你将相信写程序不会比撰写文章更难。相反，写作相对要更难一些，因此我们花费数年时间在学校学习如何写作。而通过程序模块，我们就可以解决很多令人头疼的程序编写问题。本章将从基础的 Java 开始，通过讲授程序模块，教给你各种有趣的编程知识。可能通过数周的学习，你就可以掌握用编写程序来表达自己的想法的技能。如同写作一样，一旦学会编程技能，将对你的未来产生深远的影响，并贯穿你的整个人生。

本书中你将学到的是 Java 编程语言 (Java programming language)。这个任务比学习外语要容易得多。事实上，程序语言只包含几十个常用词汇和语法规则。本书中涉及的内容也可以用 Python、C++ 或其他一些现代编程语言来表达。本书中，我们尽可能地用 Java 来讲授，以使读者尽快地学会创建和运行程序。一方面，我们聚焦于讲解如何编程，而不是如何使用 Java；另一方面，编程的部分难点就在于把握特定条件下的程序细节。使用 Java 进行编程，可以让你适应多种计算机，也可以帮助你快速的学习其他语言，如入门阶段的 C 语言和专业的 Matlab 语言。

1

1.1 你的第一个程序

在本节中，通过学习程序运行的基本步骤，我们将带领你进入 Java 编程世界。与许多你习惯使用的其他应用程序（如文字处理程序、电子邮件程序和 Web 浏览器）不同，Java 平台（以下简称 Java）是一系列应用程序的组合。与任何应用程序一样，你需要确保 Java 已正确安装在计算机上。大多数计算机预装有 Java，若没有，也可以自行下载。同时，你还需要一个文本编辑器和一个终端应用程序。接下来，登录如下网址：

<http://introcs.cs.princeton.edu/java>

可以查找到安装 Java 编程环境的说明。在后续的章节中，我们把这个网站称为本书官网 (booksite)，它涵盖本书的大量补充信息和参考资料。

Java 编程 为介绍方便，我们将程序开发划分为以下三个步骤：

- 创建 (create) 名为 MyProgram.java 的程序。
- 在终端窗口中键入 `javac MyProgram.java` 进行编译 (compile)。
- 在终端窗口中键入 `java MyProgram` 来执行 (execute) 或运行 (run) 它。

第一步，你需要在空白屏幕上键入一系列字符，就像撰写电子邮件或文章一样。程序员使用代码 (code) 来表示程序文本，使用编码 (coding) 来表示创建和编辑代码的行为。第二步，利用系统应用来编译你的程序（即将你的程序编译成更适合计算机的语言形式），并将运行结果保存至文件 `MyProgram.class`。第三步，将计算机的控制权从操作系统转移到你的程序（完成后再将控制权返回给操作系统）。我们有多种不同的方式来创建、编译和执行程序。对于小程序来说，下面我们给出的这一系列顺序是最容易描述和使用的。

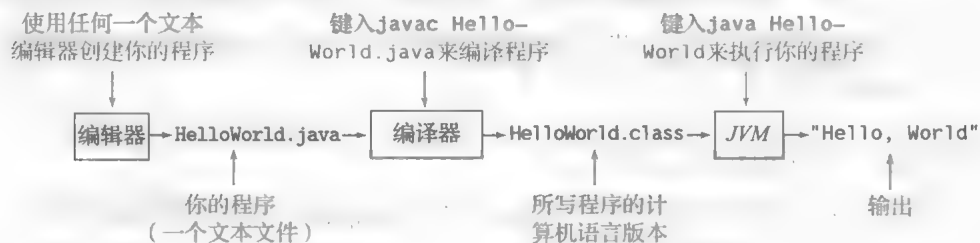
2

创建程序。Java 程序实际上只不过是一系列字符，像一段文字或者一首诗一样存储在一个扩展名为 `.java` 的文件中。因此，要创建一个 Java 程序，你只需要简单地输入该字

符序列，就像你输入电子邮件或任何其他计算机应用程序一样。你可以使用任何文本编辑器（text editor），也可以使用本书官网上推荐的更复杂的集成开发环境工具（integrated development environment）。这些工具的功能对于完成我们本书中涉及的各种程序来说绰绰有余，但它们使用起来并不困难，而且具有许多有用的特性，并在实际的软件开发中被专业人士广泛使用。

编译程序。通常，初学者认为 Java 是最便于计算机理解的编译语言，但事实上，Java 是最便于程序员理解的编译语言。计算机的语言比 Java 更原始。编译器（compiler）是将程序从 Java 语言转换为更适合在计算机上执行的语言的应用程序。编译器使用 .java 扩展名的文件作为输入（你的程序）并生成一个文件名相同但带有 .class 扩展名（计算机语言版本）的文件。要使用 Java 编译器，需要在终端窗口中键入 javac 命令，后面跟着要编译的程序的文件名即可。

执行（运行）程序。一旦程序编译成功，你就可以执行（或运行）它。这是最激动人心的时刻，你的程序将会控制你的计算机（在 Java 允许的范围之内），或者应该说让你的计算机遵循你的指示。更准确地说，Java 虚拟机（简称 JVM）命令计算机听从你的指示。若使用 JVM 执行程序，请在终端窗口中键入 java 命令，然后键入程序名称。



开发一个 Java 程序的过程

程序1.1.1 Hello, World

```

public class HelloWorld
{
    public static void main(String[] args)
    {
        // 在终端窗口中打印 "Hello, World"
        System.out.println("Hello, World");
    }
}
  
```

该代码是一个完成简单任务的Java程序，也是初学者的第一个程序。下面的方框显示了编译和执行该程序的结果。终端应用程序输出命令提示符（本书中为%。符号），并执行你键入的命令（以下示例中首先为javac，然后为java）。按照惯例，我们将输入文字高亮加粗，输出字体则为正常字体。在这个例子中，程序结果是在终端窗口中输出文本：Hello,World。

```

% javac HelloWorld.java
% java HelloWorld
Hello, World
  
```

程序 1.1.1 是一个完整的 Java 程序的例子。它的名字是 HelloWorld，它的代码应该存储在一个名为 HelloWorld.java 的文件中（这是 Java 中的文件命名惯例）。这个程序的唯一功能是将消息打印到终端窗口。为了全书的统一性，我们将使用一些标准的 Java 术语来描述程序，你现在可能还不理解这些术语是什么意思，我们将会在本书后面的章节给出它们的具体定义：程序 1.1.1 由一个名为 HelloWorld 的类（class）组成，这个类里只包含了一个名为 main() 的方法（method）（当引用文本中的方法时，我们在名称后面加上小括号“()”，这样就可以把它们与其他类型的名称区别开来）。在 2.1 节之前，我们所有的类都将具有相同的结构，因此你可以暂时将“类”理解为“程序”。

4

每一个方法的第一行定义名称和其他信息；接下来的语句（statement）序列用大括号括起来，每行以分号隔开。目前，你可以将“编程”视为“指定一个类的名称，然后给这个类的 main() 方法指定一系列语句”的过程，程序的核心就是 main() 方法中的语句序列，即它的方法体（body）。程序 1.1.1 包含两个语句：

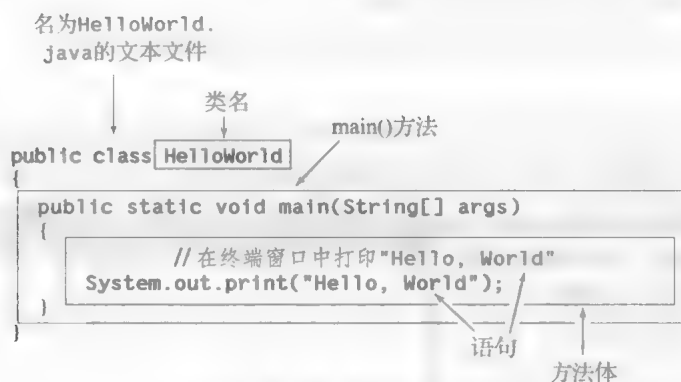
- 第一行语句是注释语句（comment），是程序的说明文档。

在 Java 中，单行注释以两个“/”字符开始，直到本行结尾。在本书中，我们将注释显示为灰色。这些注释可以供人类阅读，但 Java 会直接跳过。

- 第二个语句是打印语句（print statement）。它调用名为 System.out.println() 的方法将文本消息（一对双引号之间的字符串）打印到终端窗口。

在接下来的两节中，你将学到许多其他不同类型的语句，可以用来编写更加复杂的程序。目前，我们只使用注释和打印语句，就像 HelloWorld 程序中使用的那样。

当你在终端窗口中键入 java，后跟一个类名时，系统将调用在该类中定义的 main() 方法，并逐个执行其语句。因此，键入 java HelloWorld 会导致系统调用程序 1.1.1 中的 main() 方法并执行它的两个语句。第一个语句是 Java 忽略的注释。第二个语句将指定的消息打印到终端窗口。



程序各部分解析

5

自 20 世纪 70 年代以来，程序员在入门时编写的第一个程序就是打印“Hello, World”，这已经成为一个传统。所以，你应该将程序 1.1.1 中的代码输入文件，编译并执行。通过这个过程，你就可以跟随无数前辈的脚步学习编程。此外，你也可以顺便检查你是否具有可用的编辑器和终端应用程序。刚开始，完成在终端窗口中打印消息这样一个简单的任务可能看起来比较无趣。然而，通过打印功能，我们可以直观地看到程序在做什么，而这也恰好是编程最基本的功能之一。

从现在开始，我们编写的所有程序代码都与程序 1.1.1 基本一样，只是 `main()` 方法中的语句序列不同。因此，你不需要从零开始编写每一段程序。相反，你可以这样：

- 将 `HelloWorld.java` 复制为一个新文件，然后重新给新文件命名，扩展名为 `.java`。
- 将文件中第一行的 `HelloWorld` 替换为新的程序名称。
- 将原来程序中的注释和打印语句替换成不同的 Java 语句。

程序因程序名称和程序语句的不同有所区别。每个 Java 程序必须存储在一个文件中，而这个文件的名称必须与程序中 `class` 后的词相同，它的扩展名也必须是 `.java`。

错误。初学者很容易混淆编辑、编译和执行程序之间的区别。当你学习编程时，应该将这些过程分开，以便发生错误的时候能够清楚地找到原因。编程过程中出现错误是不可避免的。

你可以通过在创建程序时仔细检查程序来修改或避免大多数错误，就像你在编写电子邮件时仔细检查并修改拼写和语法错误一样。在编译程序时会出现一些错误，我们称之为编译时错误（`compile-time error`），因为这些错误会阻止编译器进行后续的代码翻译。对于编译完成后，在执行程序时才出现的错误，我们将其称为运行时错误（`run-time error`）。

一般来说，程序中的错误也被称为 **Bug**。**Bug** 是程序员的天敌：因为报错信息可能会令人困惑，错误的来源可能很难找到。因此，你首先要学习的是识别错误。你会学着在编码时

6 小心谨慎，避免出现这些错误。本节最后的问答环节显示了常见的 **Bug** 示例。

程序 1.1.2 使用命令行参数

```
public class UseArgument
{
    public static void main(String[] args)
    {
        System.out.print("Hi, ");
        System.out.print(args[0]);
        System.out.println(". How are you?");
    }
}
```

该程序显示了我们如何控制程序行为：在命令行中提供一个参数，程序就会把这个参数打印出来。通过参数我们能够控制程序的行为。

```
% javac UseArgument.java
% java UseArgument Alice
Hi, Alice. How are you?
% java UseArgument Bob
Hi, Bob. How are you?
```

输入和输出 通常，我们在程序中提供输入参数（`input`），程序会据此产生输出。`UseArgument`（程序 1.1.2）中展示了提供输入数据的最简单方法。每当执行程序 `UseArgument` 时，它会接受在程序名称之后键入的命令行参数，并将其作为消息的一部分打印回终端窗口。执行此程序的结果取决于你在程序名称后键入的内容。通过使用不同的命令行参数执行程序，可以生成不同的打印结果。我们将在稍后的 2.1 节中更详细地讨论用来将命令行参数传递给程序的机制。现在可以简单理解为 `args[0]` 是你在程序名称后输入的第一个命令行参数，

`args[1]` 是第二个，以此类推。因此，你可以在程序体中使用 `args[0]` 来表示你在命令行上键入的第一个字符串，`UseArgument` 程序中就是这么做的。

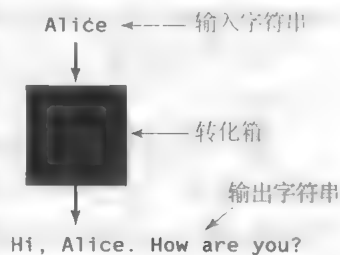
除了 `System.out.println()` 方法之外，`UseArgument` 也调用了 `System.out.print()` 方法。这个方法与 `System.out.println()` 相似，但是它只打印指定的字符串（而不再在字符串的后面加一个换行符）。

7

就像前边提到的，或许你会认为，编写一个这样的打印程序不过尔尔，但事实上，这也是程序可以实现的基本功能之一——使得用户能够控制程序的执行。`UseArgument` 代表的简单模型足够使我们思考 Java 的基本编程机制，并可解决各种有趣的计算问题。

不难发现，`UseArgument` 程序准确地将命令行参数中的字符串映射到终端窗口。使用它时，我们可能将 Java 程序视为一个转化箱，即将一段输入字符转化为输出字符。

这个模型是有吸引力的，因为它不仅简单，而且还足够普遍，原则上可以完成任何计算任务。例如，Java 编译器本身就是一个程序，它只需要一个字符串输入（.java 文件）并生成另一个字符串作为输出（相应的 .class 文件）。在后面的学习中，你将能够编写完成各种有趣任务的程序（尽管并非像编译器那么复杂的程序）。目前，我们的程序对于输入和输出数据的大小和类型还有限制。在 1.5 节中，你将看到如何将更复杂的机制用于程序的输入和输出。特别是，你将看到我们可以使用任意长的输入和输出字符串，甚至可以使用其他类型的输入数据，如声音和图片等。



Java 程序的直观图

8

问答环节

问：为什么选择 Java？

答：编写程序时，用几种常用语言写出的代码非常相似，所以选择哪种语言并不重要。我们使用 Java，因为它使用广泛，而且支持现代的抽象数据类型，并能够自动识别程序中的多种错误，因此它非常适合于编程学习。世上没有完美的语言，你以后肯定还会用其他语言来编程。

问：我相信书中各个程序均可运行且结果正确，是否还需要亲自尝试？

答：每个人都应该试着输入并运行 `HelloWorld` 程序。同时，每个人都可以在输入和运行 `UseArgument` 程序时，输入自己的个性化参数，查看结果，这有助于你对程序的理解。为减少打字工作量本书官网上展示了所有代码示例。该网站还提供有关如何在你的计算机上安装和运行 Java 的信息、部分练习的答案、网络连接以及编程时可能会用到的其他附加信息。

问：`public`、`static` 和 `void` 这几个词的含义是什么？

答：这些关键字指定 `main()` 函数的某些属性，你将在本书后面的内容中学到它们。目前，我们只需要知道这些是关键字，在代码中需要使用它们（因为它们是必需的）。

问：在程序代码中 `//`、`/*` 和 `*/` 字符的含义是什么？

答：它们表示注释，注释的相关内容会被编译器忽略。注释是 `/*` 和 `*/` 之间或 `//` 后的文字。注释是不可或缺的，因为它们能够帮助其他程序员了解你的代码，甚至帮助你在追溯代码时了解自身。限于本书排版格式的要求，我们在程序中不能大量使用注释，因此，我们只能在正文中使用相应的文字和图表来详细描述每个程序。本书官网上提供的程序代码都包含了更为详尽的注释。

9

问：Java 中使用制表符、空格和换行符的规则有哪些？

答：这些字符统称为空白字符。Java 编译器将程序文本中的所有空白都视为等效的。例如，我们可以按照下面的方式写 HelloWorld 程序：

```
public class HelloWorld { public static void main ( String
[] args) { System.out.println("Hello, World")          ; } }
```

这仍然是可以正常编译的代码。但是，当编写 Java 程序时，我们通常会遵循约定的规则控制间距和缩进，就像我们在写散文或诗歌时缩进段落和行一样。

问：引号如何使用？

答：前后引号中的字符会被精确地指定为要打印的内容。与上一问题中提出的连续空格作为空白字符不同，如果你在引号内输入了若干个连续的空格，那么输出时就会显示相应数量的空格。如果你不小心忽略了反引号，编译器可能会非常困惑，因为引号总是成对出现的，一对引号用于将字符串中的字符和程序的其他部分区分开。

问：当你漏写了一个大括号或拼写错一个单词（如 public、static、void 或 main）时，会发生什么？

答：这取决于你输入的内容。这种错误称为语法错误（syntax error），通常由编译器捕获。例如，我们编写一个与 HelloWorld 完全相同的程序 Bad，只是漏掉了第一个左大括号所在的行（并将程序名称从 HelloWorld 更改为 Bad），那么在编译时将获得以下信息：

```
% javac Bad.java
Bad.java:1: error: '{' expected
public class Bad
          ^
1 error
```

10

从消息中你可能会推测出你需要插入一个左大括号。但编译器可能无法准确地告诉你犯了什么错误，所以错误消息可能难以理解。例如，如果漏掉的是第二个左大括号而不是第一个左大括号，则会收到以下消息：

```
% javac Bad.java
Bad.java:3: error: ';' expected
    public static void main(String[] args)
                                ^
Bad.java:7: error: class, interface, or enum expected
}
^
2 errors
```

想要了解这些消息，可以使用的一种方法是故意将错误引入一个简单的程序中，然后看看会发生什么。无论错误消息是什么，你都应该把编译器当成一个朋友，因为它会告诉你程序存在什么问题。

问：还有哪些我可以使用的 Java 方法？

答：Java 的方法有几千个。从下一节开始，我们会逐步介绍它们。

问：运行 UseArgument 程序时弹出一个奇怪的错误消息。这是什么问题？

答：很可能你漏掉了一个命令行参数：

```
% java UseArgument
Hi, Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at UseArgument.main(UseArgument.java:6)
```


Java 提示你运行程序时没有按照约定键入命令行参数。你将在 1.4 节中了解有关数组索引的更多详细信息。记住这个错误消息——你可能会再次看到它。即使经验丰富的程序员也会偶尔忘记键入命令行参数。

11

练习

1.1.1 编写一个打印 10 次 “Hello, World” 消息的程序。

1.1.2 描述如果在 HelloWorld.java 中删掉以下内容，将会发生什么：

- | | |
|-----------|-----------|
| a. public | b. static |
| c. void | d. args |

1.1.3 描述如果在 HelloWorld.java 中拼错（如漏掉第二个字母）以下内容，将会发生什么：

- | | |
|-----------|-----------|
| a. public | b. static |
| c. void | d. args |

1.1.4 如果在 HelloWorld.java 的 print 语句中将双引号放在不同的行上，如下所示，描述一下会发生什么：

```
System.out.println("Hello,
                    World");
```

1.1.5 如果你尝试使用以下命令行执行 UseArgument，描述一下会发生什么：

- | | |
|-------------------------------|------------------------------|
| a. java UseArgument java | b. java UseArgument @!&^% |
| c. java UseArgument 1234 | d. java UseArgument.java Bob |
| e. java UseArgument Alice Bob | |

1.1.6 参考程序 UseArgument.java 并创建一个新程序 UseThree.java：它需要三个名字作为命令行参数，并按照给定名字相反的顺序将它们打印出来，如输入 “java UseThree Alice Bob Carol”，则打印 “Hi Carol, Bob, and Alice”。

12
13

1.2 内置数据类型

使用 Java 编程时，必须注意程序正在处理的数据类型。1.1 节中的程序处理的是字符串数据，本节中的许多程序主要处理数字数据，我们在本书后面将会研究更多其他的数据类型。了解各种数据类型之间的区别是非常重要的，因此我们必须给出数据类型的准确定义：数据类型（data type）是值和对这些值的操作的集合。例如你熟悉的数字数据，如整数和实数，以及对这些值的操作，如加法和乘法。在数学中，我们学习到数字的集合是无限的；在计算机程序中，我们能够处理的集合都是有限的。我们在这些集合上定义的操作也只是针对集合中有限个数的相关数据而设计的。

Java 中有八种基本（primitive）数据类型，大多用于表示不同类型的数字，我们最常使用的有：int 表示整数，double 表示实数，boolean 表示真 / 假值。Java 库中还包含其他的数据类型，如 1.1 节中的程序使用 String 类型表示字符串。对于 Java 输入和输出来说，String 类型较为特殊，它具有基本类型的一些特征，它的一些操作被内置在 Java 语言中。为了清楚起见，我们将基本类型和 String 类型统称为内置（built-in）类型。在本章中，我们只关注基于内置类型计算的程序。稍后，我们将学习 Java 库中的其他数据类型并构建自己的数据类型。事实上，Java 编程工作通常集中在构建数据类型上，我们将在第 3 章中详细讨论。

在定义基本术语之后，我们下面研究几个示例程序和代码片段，这些程序展示了不同类型的数据的使用。这些代码片段并没有做太多实际的计算，但是很快你就会看到基于这些代

码片段写出的更长、更复杂的程序代码。了解数据类型（包括它们的值和操作）是开始编程的重要步骤，它为我们在下节开始处理更复杂的程序做了铺垫。本节中出现的代码片段在你以后的编程工作中会经常用到。

14

类型	值的集合	常用操作符	实例
int	整数	+ - * / %	99 12 2147483647
double	浮点数	+ - * /	3.14 2.5 6.022e23
boolean	布尔值	&& !	true false
char	字符		'A' '1' '%' '\n'
String	字符串	+	"AB" "Hello" "2.5"

基本内置数据类型

术语 要谈论数据类型，我们需要介绍一些术语。如下代码片段：

```
int a, b, c;  
a = 1234;  
b = 99;  
c = a + b;
```

第一行是一个声明语句（declaration statement），它使用标识符（identifier）a、b 和 c 声明三个变量（variable），其数据类型为 int。接下来的三行是赋值语句（assignment statement），用文字常量（literal）1234 和 99 以及表达式（expression）a+b 来更改变量的值，最终 c 的值为 1333。

文字常量（literal）。文字常量用于在 Java 代码中表示数据类型的值。比如 1234 或 99 等数字表示的是 int 类型的值；添加一个小数点后，如 3.14159 或 2.71828，表示的是 double 类型的值；我们使用关键字 true 或 false 表示 boolean 类型的值（这种类型只有这两个值）；使用一对引号中包含的字符序列，如 “Hello,World”，来表示 String 类型的值。

运算符（operator）。运算符用于在 Java 代码中表示数据类型之间的操作。例如，Java 使用 “+” 和 “*” 表示整数及浮点数的加法和乘法；Java 使用 &&、|| 和 ! 代表布尔运算等。我们将在本节后面介绍内置类型中最常用的运算符。

15

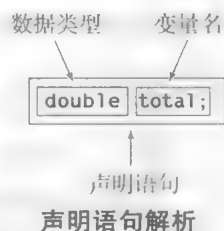
标识符（identifier）。标识符用于在 Java 代码中表示各类元素的名称，如变量、类、方法等。标识符是由一系列字母、数字、下划线和货币符号（\$）构成的，其中第一个字符不可以是数字。例如，abc、Ab\$、abc123 和 a_b 都是合法的 Java 标识符，但 Ab*、1abc 和 a+b 不是。标识符区分大小写，所以 Ab、ab 和 AB 是不同的标识符名称。特定保留字如 public、static、int、double、String、true、false 和 null 等，不能用作标识符。

变量（variable）。变量是保存数据类型值的载体。在 Java 中，每个变量都有一个特定类型，并存储该类型可能的值。例如，int 变量可以存储值 99 或 1234，但不能存储 3.14159 或 “Hello, World”。相同类型的不同变量可能存储相同的值。另外，顾名思义，变量的值可能随着计算的展开而改变。例如，我们在本书的程序中使用一个名为 sum 的变量来保存一系列数字的总和，并且随着程序运行不断累加。我们使用声明语句创建变量，并用表达式计算变量，后面我们会详细解释相关概念。

声明语句（declaration statement）。要在 Java 中创建变量，需要使用一个声明语句，简称声明。一个声明语句包括一个数据类型，后跟一个变量名。处理声明语句时，Java 会在内存中预留用于存储指定类型的数据的内存区域，并将变量名称与该内存区域相关联，以便在

以后的代码中使用变量可以访问到该值。为简单起见，Java 支持在一个声明语句中声明同一类型的多个变量。

变量命名规则 (variable naming convention)。程序员通常依照约定俗成的规则来命名变量。在本书中，我们的命名规则是，每个变量名由一个小写字母开始，后面可以有若干个小写字母、大写字母或者数字，而且变量的名称要有具体的含义。当一个变量名是由多个单词构成时，每个单词的首字母需要大写。例如，`i`、`x`、`y`、`sum`、`isLeapYear` 和 `outDegrees` 等。程序员将此命名方式称为骆驼命名法 (camel case)。

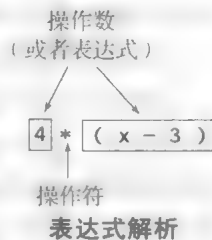


常数变量 (constant variable)。常数变量 (简称常量) 这个词看起来有点前后矛盾，我们用它来描述在执行程序过程中不会发生改变的变量 (或者说在程序的每一次执行期间它的值都一样)。在本书中，常数变量的名称通常由大写字母、数字和下划线组成，而且第一个字符需要是大写字母。例如，`SPEED_OF_LIGHT` 和 `DARK_RED`。

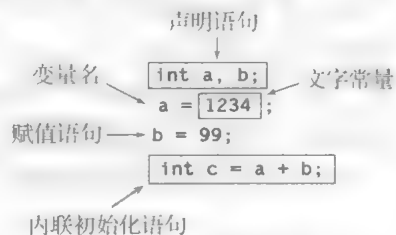
16

表达式 (expression)。表达式是由文字常量、变量和操作组成的。Java 能够识别表达式并能够计算 (evaluate) 它们的结果。对于基本数据类型，表达式通常看起来就像数学公式，使用运算符指定要对一个或多个操作数执行的数据操作。我们使用的大多数运算符是二元运算符，它们只需要两个操作数，例如 `x-3` 或 `5*x`。每个操作数可以是任意表达式，也可以包含括号。例如，我们可以写 `4*(x-3)` 或者 `5*x-6`，Java 会明白我们的意思。表达式表示一系列将要被执行的操作，代表的是这一系列运算之后得到的结果。

运算顺序 (operator precedence)。表达式中可能包含多个运算符，那么此时应该按什么顺序进行运算？Java 中定义了一套运算符的优先级规则，能够清晰地定义运算的顺序，并且使用起来非常自然。对于算术运算，在加法和减法之前先执行乘法和除法，因此 `a-b*c` 和 `a-(b*c)` 表示相同的操作顺序。当算术运算符具有相同的优先级时，考虑运算符的左结合性 (left associativity) 以确定运算顺序，因此 `a-b-c` 和 `(a-b)-c` 表示相同的操作顺序。如果想要改变顺序的话，可以使用括号来改变规则，所以对于上面的式子，如果想先计算 `b-c`，那么你需要写成 `a-(b-c)`。你将来可能会遇到一些巧妙使用优先级规则的 Java 代码，但是我们在本书中尽量避免使用这种代码，对于复杂的运算式我们会使用括号来标识运算顺序。如果你对这部分内容感兴趣，可以在本书官网上找到关于优先级规则的详细解读。



赋值语句 (assignment statement)。赋值语句用于将数据值与变量关联起来。当我们在 Java 中编写 `c=a+b` 时，我们表示的并不是一个数学等式，这个语句表示一个操作 (action)：将变量 `c` 的值设置为变量 `a` 加变量 `b`。确实，从数学角度分析，在执行赋值语句之后 `c` 的值与 `a+b` 的值是相等的，但是赋值语句的目的不是比较，而是要改变 (或初始化) `c` 的值。赋值语句的左侧必须是单个变量，右侧可以是产生数值的任意表达式，同时，赋值语句两端的数据类型必须是兼容的。所以，“`1234=a;`”和“`a+b=b+a;`”都是 Java 中的无效语句。这里需要强调的是，这里等号 (=) 的含义与数学方程式中的不同。



17

内联初始化 (inline initialization)。在表达式中使用变量之前，必须首先声明变量并为其指定初始值，否则

使用基本数据类型

会导致编译时错误。为了使编程简单，可以将声明语句与赋值语句组合在一起，这样的语句称为内联初始化语句（inline initialization statement）。例如，以下代码声明两个变量 `a` 和 `b`，并分别将它们初始化为值 1234 和 99：

```
int a = 1234;
int b = 99;
```

通常，我们在程序中首次使用某个变量时，即声明并初始化这个变量。

跟踪变量值的变化。为了检验你是否完全理解了赋值语句的使用方法，请尝试理解以下代码。注意这段代码的意思是交换 `a` 和 `b` 的值（假设 `a` 和 `b` 是 `int` 变量）：

```
int t = a;
a = b;
b = t;
```

为了分析这个过程，我们可以建立一张跟踪表（trace），研究每个语句之后的变量值的变化。这个方法也是检验程序最可靠的办法。

类型安全（type safety）。Java 需要你明确每个变量的类型，这样，Java 就能够在编译时发现类型不匹配造成的错误，并提醒程序中的潜在漏洞。例如，你不能将 `double` 值赋值给 `int` 变量，不能将 `String` 类型与布尔值相乘，也不能在表达式中使用未初始化的变量。这种情况类似于在科学计算中，需要确保参与运算的数值的单位是匹配的（例如，将以英寸[⊖]为单位的数值与以磅[⊖]为单位的数值相加是没有意义的）。

	a		b
int a, b;	未定义	未定义	
a = 1234;	1234	未定义	
b = 99;	1234	99	
int t = a;	1234	99	1234
a = b;	99	99	1234
b = t;	99	1234	1234

你的第一张变量跟踪表

接下来，我们将介绍最常使用的基本内置类型（字符串、整数、浮点数和真/假值）的详细信息，并通过示例代码说明它们的使用方法。要了解如何使用数据类型，你不仅需要了解这些类型能够表示的数值集合，还需要了解这些类型的变量可以执行的操作、调用这些操作的语言机制以及这些类型的文字常量的使用规则。

[18]

字符和字符串 字符（`char`）类型表示单个字母、数字字符或者符号，就像我们日常使用计算机时输入的那些字符一样。字符类型值最多可以有 2^{16} 种，我们常用的有字母、数字、符号和空白字符（如制表符 `tab` 和换行符）。可以通过在一对单引号内包含一个字符来指定字符类型的文字常量，例如，`'a'` 代表字母 `a`。对于制表符、换行符、反斜线、单引号和双引号，我们需要使用特殊的转义符（escape sequence）来表示，分别是 `\t`、`\n`、`\\`、`\'` 和 `\"`，这些字符使用 16 位整数表示，编码的方式称为 Unicode。除了常见的字符外，还有一些转义符用于表示键盘上找不到的特殊字符（请参阅本书官网以找到详细内容）。对于字符型变量，我们通常只进行赋值操作，一般不会执行其他操作。

字符串（`String`）类型表示一个字符序列。可以通过在一对双引号内包含一个字符序列来指定字符串类型的文字常量（例如 `"Hello, World"`）。`String` 数据类型不是基本类型，但 Java 有时会把它当成基本数据类型。例如，对于字符串类型而言，连接符（`+`）表示的是连接操作，操作数是两个字符串，运算的结果是将第二个操作数的字符附加到第一个操作数的字符后面而形成一个新的字符串类型变量。

值	字符
典型	<code>'a'</code>
文字常量	<code>'\n'</code>

Java 内置的 `char` 数据类型

⊖ 英寸是长度单位，1 英寸 = 0.0254 米。——编辑注
⊖ 磅是质量单位，1 磅约等于 0.45 公斤。——编辑注

通过定义 String 变量并在表达式和赋值语句中使用，连接操作可以完成一些特殊的计算任务。例如，Ruler（程序 1.2.1）计算一个标尺函数（ruler function）值的表格，该函数描述了标尺刻度标记的相对长度。从中可以看出，编写一个简短的程序以产生大量输出是很容易的。按照代码的规律对它进行简单的扩展，打印出第 5 行、第 6 行、第 7 行等，你可以看到每次向该程序添加两个语句时，输出的字符数量将会加倍。具体来说，如果程序打印 n 行，第 n 行将包含 2^n-1 个数字。也就是说，如果要以这种方式继续添加语句，当程序打印 30 行时，会打印超过 10 亿个数字。

值	字符序列
典型文字常量	"Hello, World"
运算	" * "
运算符	连接
	+

Java 的内置字符串数据类型

表达式	值
"Hi, " + "Bob"	"Hi, Bob"
"1" + " 2 " + "1"	"1 2 1"
"1234" + " " + " " + "99"	"1234 + 99"
"1234" + "99"	"123499"

典型字符串表达式

程序1.2.1 字符串连接

```
public class Ruler
{
    public static void main(String[] args)
    {
        String ruler1 = "1";
        String ruler2 = ruler1 + " 2 " + ruler1;
        String ruler3 = ruler2 + " 3 " + ruler2;
        String ruler4 = ruler3 + " 4 " + ruler3;
        System.out.println(ruler1);
        System.out.println(ruler2);
        System.out.println(ruler3);
        System.out.println(ruler4);
    }
}
```

该程序打印出标尺上每一个刻度的相对长度。第 n 行输出表示的是以 $\frac{1}{2^n}$ 英寸为最小刻度标尺上每个标记的相对长度。例如，第 4 行输出给出了刻度是 $\frac{1}{16}$ 英寸的标尺上刻度间隔标记的相对长度。

```
% javac Ruler.java
% java Ruler
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```



标尺函数 ($n=4$)

到目前为止，我们最常使用连接操作的地方是在 System.out.println() 函数中，将计算结果与其他信息输出结合在一起。例如，我们可以简化 UseArgument（程序 1.1.2），将它的 main() 函数中的三个语句替换为如下单个语句：

```
System.out.println("Hi, " + args[0] + ". How are you?");
```

我们首先详细地介绍了 String 类型，因为在程序中，无论我们处理的数据类型是什么，

都需要在输出（和命令行参数）中将结果转化成 `String` 类型。接下来，我们介绍在 Java 中如何将数字转换为字符串，以及如何将字符串转换为数字。

将数字转换为字符串进行输出。如本节开头所述，Java 内置的 `String` 类型有一些特殊的使用方法。其中一个特殊使用方法即你可以轻松地将任何类型的值转换为 `String` 类型值：每当我们使用“+”运算符时，如果其中一个操作数是 `String` 类型，Java 会自动将另一个操作数也转换为 `String` 类型，从而产生一个新的字符串，这个字符串中包含了第一个操作数的所有字符，后跟第二个操作数的字符。例如，以下两个代码片段的結果：

```
String a = "1234";      String a = "1234";
String b = "99";        int b = 99;
String c = a + b;        String c = a + b;
```

都是一样的：它们都会使 `c` 被赋值为“123499”。我们使用这种自动转换来形成一些 `String` 类型的值，并用在 `System.out.print()` 和 `System.out.println()` 中。例如，我们可以写出如下语句：

```
System.out.println(a + " + " + b + " = " + c);
```

如果 `a`、`b` 和 `c` 分别是值为 1234、99 和 1333 的 `int` 型变量，那么此语句打印字符串“1234+99=1333”。

将字符串转换为输入数字。Java 还可以将我们输入为命令行参数的字符串转换为基本类型的数值。为此，我们使用 Java 库函数 `Integer.parseInt()` 和 `Double.parseDouble()`。例如，在程序文本中输入 `Integer.parseInt("123")` 等效于输入整数 123。如果用户键入 123 作为第一个命令行参数，那么代码 `Integer.parseInt(args[0])` 将字符串值“123”转换为 `int` 值 123。在本节的程序中将会看到这种用法的几个示例。

通过这些机制，我们仍然可以将每个 Java 程序视为一个转化箱，它接受字符串参数并产生字符串结果，但是现在我们可以将这些字符串看作数字，并对它们进行一些计算。

[21]

整数 `int` 类型用于表示整数（自然数），它的取值范围在 -2^{31} 和 $2^{31}-1$ 之间。之所以存在这样的界限，是因为整数是以 32 位二进制数（binary digit）表示的，即整数类型变量有 2^{32} 个可能的值（术语二进制数在计算机科学中使用非常广泛，我们常常简称之为比特（bit）：一个比特的值只能是 0 或 1）。`int` 值的取值范围是不对称的，因为 0 包含在正值中。你可以在本节末尾的问答环节看到更多、更详细的有关数字表示的信息，但在当前的上下文中，你只需要理解 `int` 的取值是有限的，只能是刚刚给出的范围内的一个值。可以使用十进制数字 0~9 的序列指定一个 `int` 文字常量（这样的文字常量会被解释为十进制数，而且它表示的值应落在定义的范围內）。我们经常使用 `int` 型变量，当我们编写程序时，自然而然地就会用到它们。

表达式	值	说明
99	99	整数型文字常量
+99	99	正号
-99	-99	负号
5+3	8	加法
5-3	2	减法
5*3	15	乘法
5/3	1	取整除法 ^①
5%3	2	余数
1/0		运行时错误

典型的 `int` 表达式

表达式	值	说明
3*5-2	13	"*" 优先级更高
3+5/2	5	"/" 优先级更高
3-5-2	-4	左结合
(3-5)-2	-4	优化后的书写方式
3-(5-2)	0	消除歧义的写法

典型的 int 表达式 (续)

① 只计商，不计余数的除法。——译者注

Java 中内置了用于 int 数据类型的加 / 减 (+ 和 -)、乘法 (*)、除法 (/) 和求余 (%) 的标准算术运算符。这些运算符需要两个 int 型变量做操作数，并产生一个 int 型结果，需要注意的是，除法或求余操作的除数不可以为 0。这些操作的定义和初等教育的数学课堂上的定义是一致的，只是请记住所有结果必须是整数：给定两个 int 值 a 和 b，a/b 的值是 a 除以 b 之后运算的整数部分，忽略余数值；a%b 的值是将 a 除以 b 后获得的余数值。例如，17/3 的值为 5，17%3 的值为 2。我们从算术运算得到的 int 结果与我们在数学课堂上学到的一样，除非结果的值太大，超出了 int 的 32 位表示法，那么它将以明确的方式被截断，只保留 32 位。这种情况被称为溢出 (overflow)。一般来说，我们必须小心，以确保溢出不会使得我们的代码产生错误的结果。目前，我们将以较小数值进行计算，所以你不必担心这些问题。

取值范围	-2 ³¹ 到2 ³¹ -1之间的整数					
典型文字常量	1234 99 0 1000000					
运算	符号	加法	减法	乘法	除法	求余数
运算符	+ -	+	-	*	/	%

22

Java 的内置 int 数据类型

程序1.2.2 整数的乘法与除法

```
public class IntOps
{
    public static void main(String[] args)
    {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        int p = a * b;
        int q = a / b;
        int r = a % b;
        System.out.println(a + " * " + b + " = " + p);
        System.out.println(a + " / " + b + " = " + q);
        System.out.println(a + " % " + b + " = " + r);
        System.out.println(a + " = " + q + " * " + b + " + " + r);
    }
}
```

整数的运算内置于Java中。这个程序的大部分代码都是用来输入和输出的。实际算术操作即程序中部的几个简单语句，它们会完成计算并将值分别赋给 p、q 和 r。

```
% javac IntOps.java
% java IntOps 1234 99
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
1234 = 12 * 99 + 46
```


程序 1.2.2 展示了整数的一种基本运算（乘法、除法和求余）。它还演示了使用 `Integer.parseInt()` 将命令行上的 `String` 值转换为 `int` 值，以及使用自动类型转换将 `int` 值转换为 `String` 值。

23

另外还有三种内置类型，它们在 Java 中也表示整数，只是存储的形式不同。`long`、`short` 和 `byte` 类型的用法与 `int` 类型相同，只是它们分别使用 64 位、16 位和 8 位的二进制数表示整数，因此它们的取值范围是不同的。程序员在处理巨大的整数时应使用 `long` 类型，而使用其余类型可节省空间。你可以在本书官网上找到关于每种类型的最大值和最小值，也可以根据位数自行计算出取值范围。

浮点数 `double` 类型代表浮点数，经常应用于科学和商业计算中。浮点数在计算机内部的表示方法类似科学计数法，所以它的取值范围很大。我们常常使用浮点数来表示实数，但是需要注意它们与实数不一样！实数的个数有无数个，但是我们在任何计算机中都只能表示有限数量的浮点数。浮点数确实已经能够很好地逼近实数，所以通常情况下我们都可以应用程序中使用它们，但是需要注意一些特殊的情况：用浮点数运算时可能会产生精度不够的问题。

表达式	值
<code>3.141 + 2.0</code>	<code>5.141</code>
<code>3.141 - 2.0</code>	<code>1.141</code>
<code>3.141 / 2.0</code>	<code>1.5705</code>
<code>5.0 / 3.0</code>	<code>1.6666666666666667</code>
<code>10.0 % 3.141</code>	<code>0.577</code>
<code>1.0 / 0.0</code>	<code>Infinity</code>
<code>Math.sqrt(2.0)</code>	<code>1.4142135623730951</code>
<code>Math.sqrt(-1.0)</code>	<code>NaN</code>

典型的 `double` 表达式

`double` 类型的文字常量可以用一个带有小数点的数字序列来表示。例如，`3.14159` 就是一个 `double` 类型的文字常量，表示 π 的六位数近似值。也可以使用类似科学计数法的方式指定 `double` 类型的文字常量：`6.022e23` 表示数字 6.022×10^{23} 。与整数一样，你可以按照这些规则在编程时键入浮点数的文字常量，或者将浮点数以字符串的形式当作命令行参数输入给程序。

Java 同样为 `double` 类型定义了算术运算符 `+`、`-`、`*` 和 `/` 对应的操作，与数学课本上的定义方式保持一致。除了这些内置运算符之外，Java 的 `Math` 库中还定义了平方根函数、三角函数、对数 / 指数函数以及浮点数的其他常用函数。要在表达式中使用这些函数时，你可以直接键入函数的名称和括号中的参数。例如，`Math.sqrt(2.0)` 表示的是计算 2 的平方根的近似值，该值是一个 `double` 型值。我们将在 2.1 节中更详细地讨论这一设计背后的机制，在本节末尾有更多关于 `Math` 库的详细描述。

取值范围	实数（按照IEEE 754标准定义）			
典型文字常量	<code>3.14159</code>	<code>6.022e23</code>	<code>2.0</code>	<code>1.4142135623730951</code>
操作	加法	减法	乘法	除法
操作符	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>

Java 内置的 `double` 数据类型

24

程序1.2.3 求解一元二次方程

```
public class Quadratic
{
    public static void main(String[] args)
    {
        double b = Double.parseDouble(args[0]);
        double c = Double.parseDouble(args[1]);
        double discriminant = b*b - 4.0*c;
        double d = Math.sqrt(discriminant);
        System.out.println((-b + d) / 2.0);
        System.out.println((-b - d) / 2.0);
    }
}
```

该程序使用一元二次方程求根公式求解 x^2+bx+c ，并打印。例如， x^2-3x+2 的根是1和2，因为我们可以将方程式分解为 $(x-1)(x-2)$ ； x^2-x-1 的根是 ϕ 和 $1-\phi$ ，其中 ϕ 是黄金分割比例；而 x^2+x+1 的根不是实数。

```
% javac Quadratic.java
% java Quadratic -3.0 2.0
2.0
1.0
```

```
% java Quadratic -1.0 -1.0
1.618033988749895
-0.6180339887498949
% java Quadratic 1.0 1.0
NaN
NaN
```

使用浮点数时，你遇到的第一件事就是精度（precision）问题。例如， $5.0/2.0$ 的结果打印出来是2.5，与预期一样，但 $5.0/3.0$ 的结果会显示为1.6666666666666667。在1.5节中，我们将学习Java中对数字输出时显示的有效位数的控制机制。在此之前，我们将使用Java默认输出格式。

25

计算的结果可能会是Infinity（数字太大而无法表示）或NaN（计算结果未定义）等特殊值。如果在计算时考虑到这些值，则需要考虑大量细节，但在这个阶段你可以忽略这些复杂情形，以轻松的方式使用double来编写Java程序，用你的程序代替计算器进行各种计算。例如，程序1.2.3显示了使用double类型值和求根公式计算一元二次方程的两个根。本节末尾还有几个类似的练习。

与整数有long、short和byte多种表示形式一样，实数的另一种表示形式称为float。当对数据的精度要求不高时，程序员有时使用float来节省空间。double类型大约可以保存15位有效数字，而float型仅适用于7位有效数字的情况。本书中不使用float类型。

布尔型 boolean型表示逻辑上的真假，它只可能有两个值：true和false，这两个值也是该类型的两个文字常量。每个布尔运算的操作数都需要是布尔类型，运算的结果仍然是一个布尔值，即只能取true和false这两个值之一。布尔类型的运算非常简单，但是这些数据和运算是计算机科学的基础。

为布尔定义的最重要的操作是与（&&）、或（||）、非（!），其定义为：

值	true或false		
文字常量	true	false	
运算	逻辑与	逻辑或	逻辑非
运算符	&&		!

Java 内置的布尔数据类型

- 如果两个操作数都为true，则 $a \&\& b$ 为true；如果两者中任意一者值为false，则为false。
- 如果两个操作数都为false，则 $a||b$ 为false；如果两者任意一者值为true，则为true。

- 如果 `a` 是 `false`，则 `!a` 是 `true`；如果 `a` 为 `true`，则 `!a` 为 `false`。

为了使这些定义更直观，我们通常使用一种称为真值表（truth table）的表格，对每个运算的每种可能性进行全面说明。`not` 函数只有一个操作数，操作数的两个可能值对应的每一个值均在第二列中指定。`and` 和 `or` 函数都有两个操作数，操作数值有四种不同的情况，每种可能情况的函数值在右边两列指定。

a	!a	a	b	a && b	a b
true	false	false	false	false	false
false	true	false	true	false	true
		true	false	false	true
		true	true	true	true

26

布尔运算的真值表定义

这些运算符与括号相配合，可以形成更复杂的表达式，用于表示复杂的布尔函数。相同的功能也可以用不同形式的表达式来表示。例如，表达式 `(a && b)` 和 `!(!a || !b)` 是等价的。

a	b	a && b	!a	!b	!a !b	!(!a !b)
false	false	false	true	true	true	false
false	true	false	true	false	true	false
true	false	false	false	true	true	false
true	true	true	false	false	false	true

从真值表中可以看到 `a&& b` 和 `!(!a || !b)` 是等价的

操纵这类表达式的研究被称为布尔逻辑（Boolean logic）。它是数学的一个分支领域，也是计算机运算的基础，它在计算机硬件的设计和操作中起着至关重要的作用，也是计算机硬件理论基础的起点。在本书中，我们所研究的布尔表达式主要用来控制程序的行为。通常，我们使用布尔表达式来表示一个特定的条件，如果该表达式为真，则会执行一段预先编写好的程序代码语句；如果该表达式为 `false`，那么将执行另外一段语句。我们将会在第 1.3 节中详细介绍这一机制。

比较 有一些运算符是混合类型（mixed-type）的，它们的操作数是一种类型，而产生的结果是另一种类型。最重要的混合类型运算符是比较运算符，包括 `==`、`!=`、`<`、`<=`、`>` 和 `>=`，它们的操作数可以是任意基本数字类型，运算后产生一个布尔结果。因为我们会为每一个数据类型定义它们的运算操作，每个数据类型都有比较运算，因此这些符号每一个都代表了许多种操作（即所有类型的比较操作都使用这些运算符。——译者注）。需要说明的是，比较操作要求两个操作数是相同的类型。

判断结果是否为负值

`(b*b - 4.0*a*c) >= 0.0`

判断是否是一个世纪的开始

`(year % 100) == 0`

判断是否是合法月份

`(month >= 1) && (month <= 12)`

27

典型的比较表达式

即使没有深入了解数值存储和表示的细节，但很明显，各种类型数据的操作是完全不同的。例如，比较两个 `int` 型数字，如判断 `(2<=2)` 是否为真，这是一种运算操作；比较两个 `double` 型数字，如判断 `(2.0<=0.002e3)` 是否为真，则是另外一种完全不同的运算操作。尽管种类繁多，但每种类型的定义都非常清晰，而且可以用它们来编写包含测试条件，如

(b*b-4.0*a*c) >=0.0 的代码。在以后的编程中，你会经常用到它们。

比较运算符比算术运算符的优先级低，比布尔运算符的优先级高，因此在刚才的例子中，(b*b-4.0*a*c) >=0.0 表达式并不需要括号。而且你也可以写一个不带括号的表达式以测试 int 变量 month 的值是否在 1 到 12 之间，即 month>=1 && month<=12。不过，使用括号是更好的代码书写习惯。

比较运算以及布尔逻辑可为 Java 程序中的决策提供依据。程序 1.2.4 中展示了如何使用这些运算符，你也可以在本节末尾的练习中找到其他示例。在 1.3 节中，我们将看到如何在更复杂的程序中使用布尔表达式。

程序1.2.4 闰年

```
public class LeapYear
{
    public static void main(String[] args)
    {
        int year = Integer.parseInt(args[0]);
        boolean isLeapYear;
        isLeapYear = (year % 4 == 0);
        isLeapYear = isLeapYear && (year % 100 != 0);
        isLeapYear = isLeapYear || (year % 400 == 0);
        System.out.println(isLeapYear);
    }
}
```

该程序测试输入的整数是否是公历中的闰年年号。如果年号可以被4整除（如2004），那么就是闰年；如果还可以被100整除，则还需要被400除尽才是闰年（如2000），否则就不是（如1900）。

```
% javac LeapYear.java
% java LeapYear 2004
true
% java LeapYear 1900
false
% java LeapYear 2000
true
```

运算符	含义	true	false
==	等于	2 == 2	2 == 3
!=	不等于	3 != 2	2 != 2
<	小于	2 < 13	2 < 2
<=	小于或等于	2 <= 2	3 <= 2
>	大于	13 > 2	2 > 13
>=	大于或等于	3 >= 2	2 >= 3

比较操作示例（操作数为 int 类型，结果为布尔类型）

库方法与 API 正如我们所看到的，许多编程任务除了使用内置运算符之外，还需要使用 Java 库方法。可用的库方法的数量是巨大的。在学习编程的过程中，你将学习使用越来越多的库方法，但在初学阶段最好是将注意力控制在相对较小的范围内。在本章中，你已

经使用了一些 Java 方法，如打印的方法、用于将数据从一种类型转换为另一种类型的方法，以及用于计算数学函数的方法（Java Math 库）。在后面的章节中，你不仅会学到如何使用其他方法，还可以了解如何创建和使用自己的方法。

为方便起见，我们将使用如下表格来总结你需要知道如何使用的库方法：

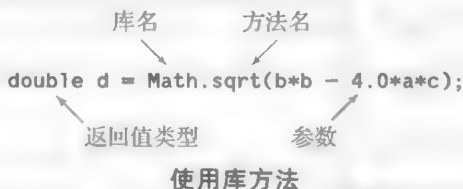
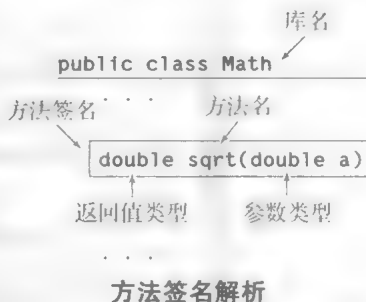
```
void System.out.print(String s)   打印s
void System.out.println(String s) 打印s并换行
void System.out.println()         换行
```

注意：任何类型的数据都可以用作参数（并将自动转换为String类型）。

打印字符串到终端的 Java 库方法

我们把这样的表叫作应用程序编程接口（Application Programming Interface, API）。在 API 表中，每个方法占一行，并描述了如何使用该方法的信息。表中的代码不是你使用该方法的代码，而是方法的签名（signature）。签名指定了方法的参数类型、方法名称和方法计算结果（返回值）的类型。

在你的代码中，你可以通过键入方法名及其参数来调用该方法，参数写在括号中，用逗号分隔。当 Java 执行你的程序时，我们称它使用给定的参数调用方法，并且该方法返回一个值。方法调用是一个表达式，所以你使用方法调用时就像使用变量和文字常量来构建更复杂的表达式一样。例如，你可以编写如 `Math.sin(x)* Math.cos(y)` 这样的表达式。参数也可以是一个表达式，所以你可以编写像 `Math.sqrt(b*b-4.0*a*c)` 这样的表达式，Java 能够理解它的意思——它会计算参数表达式，并将结果值传递给方法。



从随后的 API 表中，你可以找到 Java Math 库中的一些常用方法，也可以看到我们使用过的将文本打印到终端窗口的方法，以及将字符串转换为基本类型的 Java 方法。下表展示了这些库方法的几种调用示例。

方法调用	库	返回值类型	值
<code>Integer.parseInt("123")</code>	Integer	int	123
<code>Double.parseDouble("1.5")</code>	Double	double	1.5
<code>Math.sqrt(5.0*5.0 - 4.0*4.0)</code>	Math	double	3.0
<code>Math.log(Math.E)</code>	Math	double	1.0
<code>Math.random()</code>	Math	double	[0,1) 范围内的一个随机值
<code>Math.round(3.14159)</code>	Math	long	3
<code>Math.max(1.0, 9.0)</code>	Math	double	9.0

典型的 Java 库方法的调用


```
public class Math
```

double	abs(double a)	a 的绝对值
double	max(double a, double b)	a 和 b 中的较大者
double	min(double a, double b)	a 和 b 中的较小者

注 1: int、long 和 float 类型数据也可以使用 abs()、max() 和 min() 方法。

double	sin(double theta)	theta 的正弦值
double	cos(double theta)	theta 的余弦值
double	tan(double theta)	theta 的正切值

注 2: 角度以弧度表示 使用 toDegrees() 和 toRadians() 进行弧度角度的转换

注 3: 使用 asin()、acos() 和 atan() 计算反三角函数。

double	exp(double a)	指数操作 (e^a)
double	log(double a)	自然对数 ($\log_e a$ 或者 $\ln a$)
double	pow(double a, double b)	计算 a 的 b 次幂 (a^b)
long	round(double a)	将 a 向下取整
double	random()	[0,1) 范围内的一个随机值
double	sqrt(double a)	a 的平方根

double	E	欧拉数 e 的值 (常量)
double	PI	π 的值 (常量)

从本书官网上可以查看其他可用的方法。

Java Math 库中常用方法示例

void	System.out.print(String s)	打印 s
void	System.out.println(String s)	打印 s 并换行
void	System.out.println()	换行

用于将字符串打印到终端的 Java 库方法

int	Integer.parseInt(String s)	把 s 转换为 int 类型的值
double	Double.parseDouble(String s)	把 s 转换为 double 类型的值
long	Long.parseLong(String s)	把 s 转换为 long 类型的值

用于将字符串转换为基本类型的 Java 库方法

上文提到的方法基本都是纯方法 (pure function) —— 给出相同的参数, 它们总是会返回相同的值, 而且不会产生任何可观察到的副作用 (side effect)。只有三个方法的情况不同: Math.random() 方法不是纯方法, 因为每次调用时它都会返回一个不同的值; System.out.print() 和 System.out.println() 也不是纯方法, 因为它们产生副作用——它们会把字符串打印到终端上。在 API 中, 如果一个方法能够产生副作用, 我们使用一个动词短语来描述; 否则, 我们使用一个名词短语来描述返回值。关键字 void 表示不产生返回值的方法 (其主要目的是产生副作用)。

Math 库还定义了 Math.PI (用于表示 π) 和 Math.E (用于表示 e) 两个常量值, 可以在程序中直接使用。例如, Math.sin (Math.PI / 2) 的值为 1.0 (Math.sin() 的参数是弧度制的), Math.log (Math.E) 的值为 1.0 (Math.log() 是自然对数函数)。

这些 API 是 Java 在线文档的典型示例, 而在线文档是现代编程中的参照标准。Java

API 有大量的在线文档可供专业编程人员使用，如果你对它们感兴趣，可直接从 Java 网站或者通过本书官网找到。你不需要访问在线文档来理解本书中的代码或编写类似的代码，因为我们在文中介绍和解释了所有用到的这些库方法，并且在书末对它们做了总结。在第 2 章和第 3 章中，你将了解如何开发一些供自己使用的方法并定义自己的 API。

类型转换 现代编程的主要规则之一是你应该始终了解程序正在处理的数据类型。只有通过数据类型，你才能准确地知道每个变量可以存储哪些值、可以使用哪些文字常量，以及可以执行哪些操作。例如，假设你希望计算四个整数如 1、2、3 和 4 的平均值，自然地，你会想到表达式 $(1+2+3+4)/4$ ，但是由于类型转换约定，它产生的是 `int` 值 2 而不是 `double` 值 2.5。问题源于操作数是 `int` 值，而结果需要 `double` 数据类型，因此在某些时候需要从 `int` 转换为 `double`。在 Java 中有多种实现数据类型转换的方法。

隐式类型转换 (implicit conversion)。即使预期的结果是 `double` 值，你也可以使用 `int` 值，因为 Java 会适当地将整数自动转换为实数值。例如，“ $11*0.25$ ”的计算结果为 2.75，这是由于 0.25 是一个 `double` 类型的数字，而两个操作数必须是同一类型，因此 11 被转换为 `double` 型，而两个 `double` 类型的操作数的结果是 `double` 型。另一个例子如 `Math.sqrt(4)` 结果为 2.0，因为 `Math.sqrt()` 预期接收的参数是 `double` 类型，因此 4 被自动转换为 `double` 类型，然后它返回一个 `double` 值。这种转换称为自动升级 (automatic promotion) 或自动转换 (coercion)。在这些使用场景中，编程人员的意图很明确，而且类型转换也不会损失信息，因此系统进行自动类型转换是非常合理的。相反，如果类型转换可能会导致信息丢失（例如，将一个 `double` 值赋给 `int` 变量），那么 Java 会产生一个编译时错误。

显式转型 (explicit cast)。当意识到可能会发生信息丢失时，你可以使用 Java 内置的基本数据类型转换方法。在进行显式转型时，你必须使用转型 (cast) 操作：表达式的前面使用一对括号，在括号内填写所需转换的类型名称，就可以将表达式从原有的基本类型转换为目标类型。例如，表达式 “`(int)2.71828`” 能够实现从 `double` 到 `int` 的转换，它的结果是一个值为 2 的 `int` 型变量。在类型转换的过程中，Java 定义了合理的方式以丢弃信息（详细的处理方式请参阅本书官网）。例如，将浮点数转换为整数时会舍弃小数部分。`RandomInt`（程序 1.2.5）是一个在实际计算中使用类型转换的例子。

表达式	表达式类型	表达式值
<code>(1 + 2 + 3 + 4) / 4.0</code>	<code>double</code>	2.5
<code>Math.sqrt(4)</code>	<code>double</code>	2.0
<code>"1234" + 99</code>	<code>String</code>	"123499"
<code>11 * 0.25</code>	<code>double</code>	2.75
<code>(int) 11 * 0.25</code>	<code>double</code>	2.75
<code>11 * (int) 0.25</code>	<code>int</code>	0
<code>(int) (11 * 0.25)</code>	<code>int</code>	2
<code>(int) 2.71828</code>	<code>int</code>	2
<code>Math.round(2.71828)</code>	<code>long</code>	3
<code>(int) Math.round(2.71828)</code>	<code>int</code>	3
<code>Integer.parseInt("1234")</code>	<code>int</code>	1234

典型的类型转换

转型操作的优先级高于算术运算，即转型操作仅应用于紧随其后的值。例如，对于代码 “int value= (int) 11*0.25”，类型转换没有什么作用，因为 11 已经是一个整数，所以在它前面的 (int) 没有任何效果。在此示例中，编译器会产生错误信息，提示存在精度丢失的风险，因为乘法的计算结果为 (2.75)，将它赋值给 value 时会自动转换为 int 类型，这个过程将导致精度损失。对于程序员而言，此代码的预期计算结果可能是 (int)(11*0.25)，其值为 2，而不是 2.75。

33

程序1.2.5 通过类型转换得到一个随机整数

```
public class RandomInt
{
    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        double r = Math.random(); // 返回结果在0.0和1.0之间
        int value = (int) (r * n); // 返回结果在0和n-1之间
        System.out.println(value);
    }
}
```

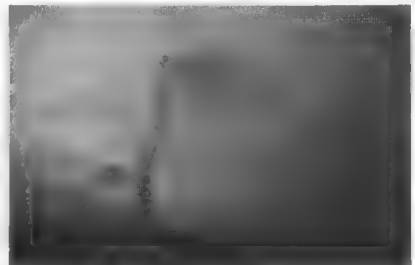
该程序使用Java方法Math.random()生成0.0（含）和1.0（不含）之间的随机数r，然后将r乘以命令行参数n以获得大于或等于0并小于n的随机数，然后使用类型转换将结果截断为0到n-1之间的整数值。

```
% javac RandomInt.java
% java RandomInt 1000
548
% java RandomInt 1000
141
% java RandomInt 1000000
135032
```

显式类型转换（explicit type conversion）。Java 中可以以以一种数据类型为参数，结果形成另外一种数据类型。我们已经使用过 Integer.parseInt() 和 Double.parseDouble() 库方法，它们分别能够将 String 值转换为 int 和 double 值。还有许多其他方法可以用于类型之间的转换。例如，库方法 Math.round() 的参数是 double 类型，经过四舍五入，返回一个 long 类型的结果。因此，Math.round(3.14159) 和 Math.round(2.71828) 的结果都是 long 类型值 3。如果你要将 Math.round() 的结果转换为一个 int，则必须使用显式转型操作。

34

最开始，程序员认为处理类型转换令人烦恼，但有经验的程序员知道，关注数据类型是编程成功的关键，也可能是避免失败的关键。在 1985 年那次著名的事件（阿丽亚娜 5 号火箭爆炸事故）中，法国火箭由于类型转换问题导致在空中爆炸。虽然你的程序中的错误可能不会引起爆炸，但是请你花点时间了解所有类型的转换。在编写了几个程序之后，你将看到对数据类型的理解不仅可以帮助你编写紧凑的代码，还能使你的意图更明确，从而避免程序



图片提供：ESA

阿丽亚娜 5 号火箭爆炸事故

中的细微错误。

总结 数据类型是一组值和基于这些值的操作。Java 中有八种基本数据类型：boolean、char、byte、short、int、long、float 和 double。在 Java 代码中，我们使用运算符和表达式来调用与每个类型相关联的操作，这些操作的使用规则与我们所熟悉的数学运算规则相同。boolean 类型的数值只有逻辑值 true 和 false，对应的操作是逻辑运算；char 类型用于表示我们键入的字符的集合；而其他六种数值类型用于进行数值计算。在本书中，我们会经常使用 boolean、int 和 double 类型；几乎不使用 short 或 float 类型。我们也经常使用另一种数据类型——String 类型，它虽然不是基本数据类型，但 Java 为之提供了与基本类型类似的一些工具方法。

当使用 Java 编程时，我们必须意识到，每个操作仅在其相应的数据类型的上下文中有意义，因此我们可能需要进行类型转换。所有类型都只能有有限数量的值，因此我们获得的结果可能不精确。

布尔类型及其操作 &&、|| 和 ! 是 Java 程序中逻辑决策的基础，它们经常与比较运算符(==、!=、<、>、<= 和 >=) 联合使用。具体来说，我们使用布尔表达式来控制 Java 的条件 (if) 和循环 (for 和 while) 结构，相关内容我们将在下一节中详细介绍。

[35]

数值类型和 Java 库使我们能够将 Java 用作一个功能强大的数学计算器。我们可以使用内置的运算符 +、-、*、/ 和 %，并配合 Math 库中的 Java 方法，来编写复杂的算术表达式。

虽然以后续内容的标准来看本节中的程序是非常简单的，但本节中的这类程序非常有用。你将在 Java 编程中广泛使用基本类型和基本数学函数，因此你现在花时间理解它们一定是值得的。

[36]

问答环节

1. 字符串

问：Java 如何在内部存储字符串？

答：Unicode 是用于编码文本的现代标准，字符串是使用 Unicode 编码的字符序列。Unicode 支持超过 100 000 个不同的字符，包括超过 100 种不同的语言，以及数学和音乐符号。

问：可以使用 “<” 和 “>” 来比较 String 值吗？

答：不可以。这些运算符仅用于基本类型值。

问：“==” 和 “!=” 呢？

答：可以，但结果可能与你预期的不一样。以上运算符对于非基本数据类型的含义不同。String 类型的变量和变量的值之间是有区别的。假设 x 是值为 “c” 的字符串时，表达式 “abc”==“ab”+x 的结果为 false，因为两个操作数存储在内存中的不同位置（即使它们具有相同的值）。这个区别是至关重要的，所以我们将会在 3.1 节中更详细地讨论这个区别，到那时你将会明白为什么会这样。

问：如何比较两个字符串，如查找书中的索引或从字典中查找单词？

答：我们将在 3.1 节介绍面向对象编程时讨论 String 数据类型和相关方法。在此之前，String 连接操作就足够用了。

问：如果需要输入的字符串文字常量太长，一行显示不完整该怎么办？

答：不可以一次输入过长的字符串常量。如果需要，应将字符串文字常量划分为独立的字符串文字并将它们连接在一起，如以下示例所示：

问：为什么 10^6 的值不是 1000000，而是 12？

答：^ 运算符不是指数运算符，它是逐位异或运算符。如果你想计算 1000000，你可以输入 1e6，也可以使用 `Math.pow(10, 6)`。需要说明的是，使用 `pow` 方法输入一个 10 的已知幂次会浪费计算资源。（因为对于已知的幂次可以输入为文字常量，而 `pow` 方法会转换为方法调用，需要运算才能得到。——译者注）

[39]

3. 浮点数

问：为什么实数的类型被命名为 `double`？

答：在表示实数时，小数点可以在构成实数的数字之间“浮动”。相反，在表示整数时，小数点被固定在最后一位有效数字的后面，而且我们并不刻意把它显示出来。（因此，实数又被称为浮点数 `float`，而 `double` 是双重精度的 `float`。——译者注）

问：Java 如何在内部存储浮点数？

答：Java 的浮点数存储遵循 IEEE 754 标准，这也是大多数现代计算机系统硬件支持的标准。该标准规定一个浮点数需使用符号位、尾数和指数共同来定义。如果你有兴趣，请参阅本书官网了解更多详情。IEEE 754 标准还规定了特殊的 `float` 值——正零、负零、正无穷大、负无穷大，以及 NaN（不是数字）。特别需要说明的是，浮点运算不会导致运行时异常。例如，表达式 `-0.0/3.0` 计算为 `-0.0`，表达式 `1.0/0.0` 计算为正无穷大，`Math.sqrt(-2.0)` 计算为 NaN。

问：浮点数有 15 位有效数字，这对我来说似乎足够了。我真的需要关心精度吗？

答：是的，因为你所习惯的实数和数学是基于无限精度的，而计算机只能处理有限的近似。例如，表达式 `(0.1+0.1==0.2)` 计算为真，但表达式 `(0.1+0.1+0.1==0.3)` 计算结果为假！像这样的陷阱在科学计算程序中并不少见，因此新手程序员应避免将两个浮点数进行比较。

问：如何初始化 `double` 变量为 NaN 或无穷大？

答：Java 具有可用于此目的的内置常量：`Double.NaN`、`Double.POSITIVE_INFINITY` 和 `Double.NEGATIVE_INFINITY`。

[40]

问：Java 的 `Math` 函数库中是否有其他三角函数的功能，如余割、正割和余切？

答：没有。但你可以使用 `Math.sin()`、`Math.cos()` 和 `Math.tan()` 来计算它们。Java 面向众多用户，但每个用户所需要的方法不同，因此，如何选择 API 集合就需要找到一个合理的折中。如果要求 API 集合中包含每一个可能使用到的方法，那么这个集合就会很大，查找所需要的函数的过程就会很痛苦。例如，你可以使用 `Math.sin(x)/Math.cos(x)` 计算正切值，而 API 中同时还提供了 `Math.tan(x)`。

问：打印 `double` 类型数字时显示出全部小数是烦人的，是否可以使用 `System.out.println()` 函数只打印小数点后两位或者后三位？

答：要实现这个功能，需要深入研究将 `double` 转换为 `String` 的方法。Java 库函数 `System.out.printf()` 提供了一个基础打印功能，C 语言和许多现代语言中也使用这一方法，我们会在 1.5 节详细讲解。在此之前，我们只能把所有数字都打印出来（这也并非全是坏事，因为这样有助于我们记住这些基本类型的差异）。

[41]

4. 变量和表达式

问：如果忘记声明一个变量会怎么样？

答：在表达式中引用该变量时，编译器会报错。例如，假设 `IntOpsBad` 是与程序 1.2.2 完全相同的一段代码，只是去掉了变量 `p` 的声明（原来被声明为 `int` 类型）。

```
% javac IntOpsBad.java
IntOpsBad.java:7: error: cannot find symbol
    p = a * b;
    ^
symbol:   variable p
location: class IntOpsBad
IntOpsBad.java:10: error: cannot find symbol
    System.out.println(a + " * " + b + " = " + p);
                                ^
symbol:   variable p
location: class IntOpsBad
2 errors
```

编译器会报告两个错误，但实际上只有一个：p 的声明被漏掉了。如果忘记声明经常使用的变量，则会收到一系列错误消息。最好的解决办法是修改第一个错误，有时后面的错误会因为第一个错误被修改而被同时修改。

问：如果忘记初始化一个变量会怎么样？

答：编译器会检查这种错误，如果你在初始化之前尝试使用表达式中的变量，则会给你一个变量未被初始化的错误消息。

问：= 和 == 运算符之间有区别吗？

答：有，它们有很大的不同！第一个是改变变量值的赋值运算符，第二个是生成布尔结果的比较运算符。你能否正确地回答这个问题，取决于你是否真正理解了本节中关于这部分的内容。

42

问：double 型变量可以与 int 型变量进行比较吗？

答：不进行类型转换则无法比较，但要说明的是，Java 通常会自动进行必要的类型转换。例如，如果 x 是 int 型变量且值为 3，则表达式 (x<3.1) 为真，因为在执行比较之前，Java 会将 x 转换为 double 型（因为 3.1 是 double 型常量）。

问：语句 “a=b=c=17;” 是否将 17 赋值给三个整数变量 a、b 和 c？

答：是的。Java 中的赋值语句是一个表达式（这个表达式的计算结果是赋值符右边的值），赋值操作符是右结合的（因此，这个表达式可以结合为 a=(b=(c=17))），其中 c=17 的运算结果就是 17，以此类推，三个变量都被赋值为 17。——译者注）。但是，这并不是是一种很好的编程风格，我们在本书中不使用这样的连续赋值（chained assignment）。

问：表达式 (a<b<c) 可以测试三个整数变量 a、b 和 c 的值是否严格按照升序排列吗？

答：不可以。这个表达式无法通过编译，因为表达式 a<b 会产生一个布尔值，然后将它与一个 int 值进行比较。Java 不支持连续比较（chained comparison）。如果想实现连续比较的功能，你需要写成 (a<b && b<c)。

问：为什么我们用 (a && b) 而不是 (a & b)？

答：Java 也有一个 & 运算符，在更复杂的高级编程课程中你可能会遇到。

问：Math.round(6.022e23) 的值是多少？

答：你应该习惯于编写一个 Java 小程序来自己寻找这些问题的答案（并思考程序产生这个结果的原因）。

问：我听说 Java 被称为静态类型语言（statically typed language）。这是什么意思？

答：静态类型意味着每个变量和表达式的类型在编译时是已知的。Java 也在编译时验证和保障类型约束。例如，如果你试图用 int 类型的变量存储 double 类型的值，或者使用 String 类型的变量作为参数调用 Math.sqrt()，程序都会在编译时遇到错误。

43

练习

1.2.1 假设 *a* 和 *b* 是整型变量。下面语句的运行结果是？

```
int t = a; b = t; a = b;
```

1.2.2 编写一个程序，使用库函数 `Math.sin()` 和 `Math.cos()` 来检查 $\cos^2\theta + \sin^2\theta$ 的值是否约等于 1，其中 θ 是由命令行参数输入的任意值。把计算的数值打印出来。为什么这些值不总是等于 1？

1.2.3 假设 *a* 和 *b* 是布尔变量。证明表达式 `(!(a && b) && (a || b)) || ((a && b) || !(a || b))` 的计算结果为 `true`。

1.2.4 假设 *a* 和 *b* 是整型变量。简化以下表达式：`(!(a < b) && !(a > b))`。

1.2.5 布尔运算的异或运算符 `^` 的定义是：如果两个操作数不同则结果为真；如果相同则结果为假。写出这个函数的真值表。

1.2.6 为什么 `10/3` 的值为 3，而不是 3.333333333？

答案：由于 10 和 3 都是整数，所以 Java 认为不需要进行类型转换，可以直接使用整数除法。如果你想让数字按照 `double` 类型处理，你应该写成 `10.0/3.0`。同样的道理，`10/3.0` 或 `10.0/3` 也会得到相同的结果，因为 Java 会进行隐式转换。

1.2.7 以下各项会打印出什么结果？

- | | |
|---|---|
| a. <code>System.out.println(2+"bc");</code> | b. <code>System.out.println(2+3+"bc");</code> |
| c. <code>System.out.println((2+3)+"bc");</code> | d. <code>System.out.println("bc"+(2+3));</code> |
| e. <code>System.out.println("bc"+2+3);</code> | |

并解释其原因。

44 1.2.8 如何利用程序 1.2.3 求一个数的平方根。

1.2.9 以下每项打印结果是什么？

- | | |
|---|---|
| a. <code>System.out.println('b');</code> | b. <code>System.out.println('b'+ 'c');</code> |
| c. <code>System.out.println((char) ('a'+4));</code> | |

并解释其原因。

1.2.10 假设变量 *a* 的声明如下：`int a = 2147483647`（这个值就是 `Integer.MAX_VALUE`）。以下各项打印结果是什么？

- | | |
|--|---|
| a. <code>System.out.println(a);</code> | b. <code>System.out.println(a+1);</code> |
| c. <code>System.out.println(2-a);</code> | d. <code>System.out.println(-2-a);</code> |
| e. <code>System.out.println(2*a);</code> | f. <code>System.out.println(4*a);</code> |

并解释其原因。

1.2.11 假设变量 *a* 的声明如下：`double a = 3.14159`。以下各项打印结果是什么？

- | | |
|--|--|
| a. <code>System.out.println(a);</code> | b. <code>System.out.println(a+1);</code> |
| c. <code>System.out.println(8/(int) a);</code> | d. <code>System.out.println(8/a);</code> |
| e. <code>System.out.println((int) (8/a));</code> | |

并解释其原因。

1.2.12 描述在程序 1.2.3 中使用 `sqrt` 而不是 `Math.sqrt` 会发生什么情况。

1.2.13 求表达式 `(Math.sqrt(2)* Math.sqrt(2) == 2)` 的值。

1.2.14 编写一个程序，该程序将两个正整数作为命令行参数，如果其中一个能将另一个整除，则打印结果 `true`。

1.2.15 编写一个程序，它将三个正整数作为命令行参数，如果其中任何一个大于或等于另外两个的和，则输出 `false`，否则输出 `true`。（注意：这个计算用于测试三个数字是否可以是个三角形的边长。）

1.2.16 假设物理系学生想在程序中计算公式 $F=Gm_1m_2/r^2$ ，他写出如下代码：

```
double force = G * mass1 * mass2 / r * r;
```

这个表达式会产生意想不到的结果。解释出现的问题并更正代码。

1.2.17 在执行以下每个语句之后，给出变量 `a` 的值：

```
int a = 1;      boolean a = true;    int a = 2;
a = a + a;      a = !a;              a = a * a;
a = a + a;      a = !a;              a = a * a;
a = a + a;      a = !a;              a = a * a;
```

1.2.18 编写一个程序，该程序需要两个浮点命令行参数 `x` 和 `y`，并打印从点 (x, y) 到原点 $(0, 0)$ 的欧几里得距离。

1.2.19 编写一个程序，该程序需要两个整数型命令行参数 `a` 和 `b`，打印一个 `a` 和 `b` 之间（含 `a` 和 `b`）的随机整数。

1.2.20 编写一个程序，打印 1 和 6 之间的两个随机整数的和（计算结果就是掷骰子时可能得到的值）。

1.2.21 编写一个程序，命令行参数为 `double` 型变量 `t`，输出 $\sin(2t) + \sin(3t)$ 的值。

1.2.22 编写一个程序，该程序需要三个 `double` 型命令行参数 `x0`、`v0` 和 `t`，并输出 $x_0 + v_0 t - gt^2/2$ 的值，其中 `g` 是常数 9.80665。（注意：这个值表示的是一个物体在 `x0` 位置按照 `v0` 的初始速度竖直向上抛出经过 `t` 秒之后的位移，单位是米。）

1.2.23 编写一个程序，该程序需要两个整型命令行参数 `m` 和 `d`，如果以这两个值作为日期，即 `m` 月 `d` 日在 3 月 20 日和 6 月 20 日之间，则输出 `true`，否则返回 `false`。

46

创新练习

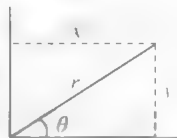
1.2.24 连续复利。假设投资 `P` 美元，年利率为 `r`（计算复利）（计算复利是指利息部分仍参与计息。——译者注）。计算 `t` 年后的资产金额。结果可以由公式 Pe^{rt} 算出。

1.2.25 风寒指数。国家气象部门对风寒指数的定义如下式，其中 `T` 为温度（华氏度），`v` 为风速（英里[⊖]每小时）。

$$w = 35.74 + 0.6215 T + (0.4275 T - 35.75) v^{0.16}$$

写一个程序，输入两个 `double` 型命令行参数，分别表示温度和速度，计算风寒指数。使用 `Math.pow(a, b)` 来计算 a^b 。注意：如果 `T` 的绝对值大于 50，或者 `v` 大于 120 或者小于 3 公式是无效的，为了简单起见，可以假设你得到的输入值一定是在这个范围内的。

1.2.26 极坐标。编写程序将笛卡儿坐标转换为极坐标。你的程序应该假设两个 `double` 型命令行参数 `x` 和 `y`，并输出极坐标 `r` 和 `θ`。使用 `Math.atan2(y, x)` 方法计算 `y/x` 的反正切值，其值在 $-\pi$ 到 π 的范围内。



极坐标

1.2.27 高斯随机数。编写一个程序 `RandomGaussian`，用于显示一个服从高斯分布的随机数 `r`。实现这一功能的一种有效方法是使用 Box-Muller 公式：

$$r = \sin(2\pi v) (-2 \ln u)^{\frac{1}{2}}$$

其中 `u` 和 `v` 是由 `Math.random()` 方法随机生成的 0 到 1 之间的实数。

1.2.28 顺序检测。编写一个程序，该程序需要输入三个 `double` 型的命令行参数 `x`、`y` 和 `z`，如果值严格上升或下降 ($x < y < z$ 或 $x > y > z$)，则输出 `true`，否则输出 `false`。

47

⊖ 1 英里 = 1609.344 米。——编辑注

- 1.2.29 星期几。编写程序，将日期作为输入，并输出当天是星期几。你的程序应该假设三个 int 命令行参数：m（月）、d（日）和 y（年）。对于 m，1 代表一月，2 代表二月，以此类推。对于输出，0 为星期天，1 为星期一，2 为星期二，以此类推。你可以使用以下公式：

$$y_0 = y - (14 - m) / 12$$

$$x = y_0 + y_0 / 4 - y_0 / 100 + y_0 / 400$$

$$m_0 = m + 12 \times ((14 - m) / 12) - 2$$

$$d_0 = (d + x + (31 \times m_0) / 12) \% 7$$

例：2000 年 2 月 14 日是星期几？

$$y_0 = 2000 - 1 = 1999$$

$$x = 1999 + 1999 / 4 - 1999 / 100 + 1999 / 400 = 2483$$

$$m_0 = 2 + 12 \times 1 = 12$$

$$d_0 = (14 + 2483 + (31 \times 12) / 12) \% 7 = 2500 \% 7 = 1$$

答：星期一。

- 1.2.30 均匀分布随机数。写一个程序，打印 0 和 1 之间的 5 个均匀分布随机数，并计算它们的平均值、最小值和最大值。你可以使用 Math.random()、Math.min() 和 Math.max() 等库函数。

- 1.2.31 墨卡托投影。墨卡托投影是一个保形（能够保留角度）投影法，经常用于将球坐标（纬度 ϕ 和经度 λ ）映射到直角坐标 (x, y) 。该方法常用于处理海图和地图。投影由方程 $x = \lambda - \lambda_0$, $y = \frac{1}{2} \ln((1 + \sin \phi) / (1 - \sin \phi))$ 定义，其中 λ_0 是地图中心点的经度。编写一个程序，从命令行获取 λ_0 和点的经纬度并打印其投影。

- 1.2.32 颜色转换。表示颜色的数据格式有多种。例如，在 RGB 格式中，使用三个整数分别表示红（R）、绿（G）和蓝（B）的级别，整数的取值范围是从 0 到 255。这种格式主要用于 LCD 显示器、数码相机和网页配色。出版书籍和杂志主要使用的格式是 CMYK 格式，使用 4 个实数分别表示青色（C）、品红色（M）、黄色（Y）和黑色（K）的等级，实数的取值范围是从 0.0 到 1.0。编写一个程序 RGBtoCMYK，将 RGB 转换为 CMYK。从命令行取三个整数 r 、 g 和 b 并打印出等价的 CMYK 值。如果 RGB 值全部为 0，则 CMY 值全部为 0，K 值为 1；否则，使用下面这些公式：

$$w = \max(r/255, g/255, b/255)$$

$$c = (w - (r/255)) / w$$

$$m = (w - (g/255)) / w$$

$$y = (w - (b/255)) / w$$

$$k = 1 - w$$

- 1.2.33 大圆（大圆是指过球心的平面和球面的交线，球面上两点的最小距离为经过两点的大圆的劣弧。航海与航空中利用这一原理而设置了大圆航线。——译者注）。编写 GreatCircle 程序，它需要四个 double 型命令行参数，分别是 x_1 、 y_1 、 x_2 和 y_2 ，用于表示地球上两点的纬度和经度，以度为单位，计算它们之间的大圆距离。大圆距离（海里）由下式给出：

$$d = 60 \arccos(\sin(x_1) \sin(x_2) + \cos(x_1) \cos(x_2) \cos(y_1 - y_2))$$

请注意，此公式使用度数，而 Java 的三角函数使用弧度。你可以使用 Math.toRadians() 和 Math.toDegrees() 在两者之间进行转换。使用你的程序计算巴黎（48.87° N，-2.33° W）和旧金山（37.8° N，122.4° W）之间的大圆距离。

1.2.34 三数字排序。编写一个程序，该程序需要输入三个整数型命令行参数，按照升序排列这三个数字并打印。你可以使用 `Math.min()` 和 `Math.max()` 库函数。

1.2.35 龙形曲线。编写一个程序，用于输出绘制龙形曲线的指令。绘制指令是 F、L 和 R 构成的字符串，其中 F 表示“向前画出 1 个单位的直线”，L 表示“向左转”，R 表示“向右转”。要得到一个 n 阶的龙形曲线，你可以将纸条对折 $n/2$ 次后，再将它展开并把每一个折痕展开为直角。解决问题的关键是要注意，阶数 n 的龙形曲线是一个阶数为 $n-1$ 的龙形曲线，后面跟着一个 L，后面跟着一个反序的阶数 $n-1$ 的龙形曲线，反序的龙形曲线的绘制方法与此类似。



阶数为 0、1、2 和 3 的龙形曲线

1.3 条件语句与循环语句

在我们之前已经研究过的程序中，每个语句都按照给定的顺序执行一次。事实时，大多数程序都比那些程序复杂得多，在语句的顺序和执行的次数上都会有变化。我们使用术语控制流来表示程序中语句的执行顺序。在本节中，我们将介绍一些语句，这些语句能够根据程序中某些变量值的逻辑来改变控制流，是编程的重要组成部分。

具体而言，我们将学习 Java 中的条件语句，即控制其他某些语句可能会或者不会被执行的语句，具体的控制标准取决于某些特定的条件。我们还会学习循环语句，在循环语句中，一些语句可能会执行多次，执行的次数也取决于特定的条件。正如你将在本节看到的那样，条件和循环语句真正利用了计算机的能力，并且使你通过编写程序来完成各种各样的任务，而这些任务在没有计算机的情况下是无法想象的。

if 语句 大多数的计算需要根据不同的输入采用不同的方式去进行。Java 中表达这些变化的一种方法是 if 语句，其格式为：

```
if (<布尔表达式>){<语句>}
```

在这里，我们使用了一个被称为模板（template）的形式符号来描述 Java 语句的结构和格式。在模板中，我们在尖括号（<>）中填入一个已经定义好的结构，用于表示我们可以在这个位置填入该类型的语句。在这个例子中，<布尔表达式>表示这个位置可以填入任何一个运算结果为布尔值的表达式，比如一个调用比较操作的表达式，<语句>表示一个语句块（一系列 Java 语句）。你已经见到过<语句>这样的结构：`main()`的主体就是这样这样一个语句块。如果语句块中只有单个语句，则花括号是可以省略的。我们可以对<布尔表达式>和<语句>进行更加严格的形式化定义，但是我们不需要深入探究这个细节层次。if 语句的含义非常明显：当且仅当表达式为真时才会执行语句块中的语句。

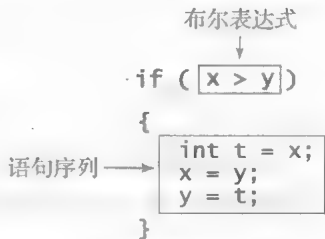
举一个简单的例子，假设你想计算出一个 `int` 值 `x` 的绝对值，具体的做法是这样的：

```
if (x < 0) x = -x;
```

（更确切地说，这段程序的功能是用 `x` 的绝对值来替换 `x` 的值。）

作为第二个简单的例子，考虑下面的语句：

```
if (x > y)
{
    int t = x;
    x = y;
    y = t;
}
```



一个 if 语句的解析

此段代码的功能是：如果需要的话，交换两个变量中的值，将两个 int 值中较小的一个放在 x 中，两个值中较大的一个放在 y 中。

你还可以在 if 语句后添加一个 else 子句，这样形成的代码用于表示选择执行两个语句（或语句序列）中的一个，具体执行哪一个取决于布尔表达式是 true 还是 false，代码模板如下所示：

```
if (<布尔表达式>) <语句T>
else               <语句F>
```

举一个需要 else 语句的简单例子，考虑以下代码，该语句将两个 int 值中最大的那个赋值给变量 max：

```
if (x > y) max = x;
else      max = y;
```

控制流程的一种可视化展示方法是流程图。流程图中的路径对应于程序中的执行流程路径。在计算的早期，当程序员使用低级语言和难以理解的控制流程时，流程图是编程的一个重要部分。在现代编程语言中，我们使用流程图来展示 if 语句等基本语句块之间的结构。



51 流程图示例 (if 语句)

下面的表格包含了使用 if 和 if-else 语句的例子。这些示例是完成简单计算功能的典型代码，你在编写程序中可能会用到。条件语句在编程中是很重要的部分。由于这些语句的语义（即意思）和它们所对应的自然语言的短语很相似，因此你可以快速地学会使用它们。

程序 1.3.1 是另一个使用 if-else 语句的例子。在这个例子中，我们模拟了硬币翻转的场景。程序的主体部分只有一条语句，语句的用法和表格例子所示的非常类似，但是需要注意的是，这个程序引入了一个值得思考的哲学问题：计算机程序能产生一个随机（random）的数吗？当然不能，但是计算机程序可以产生一些具有很多随机数特性的数字。

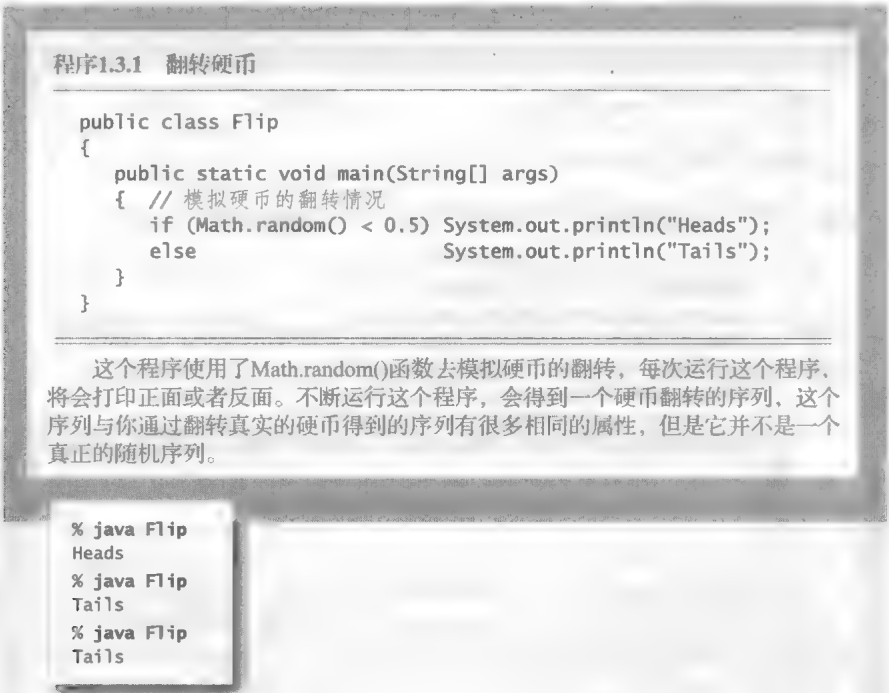
绝对值	<pre>if (x < 0) x = -x;</pre>
将较小的值放在 x 中， 将较大的值放到 y 中	<pre>if (x > y) { int t = x; x = y; y = t; }</pre>
x 和 y 中的较大值	<pre>if (x > y) max = x; else max = y;</pre>
除法的错误检查	<pre>if (den == 0) System.out.println("Division by zero"); else System.out.println("Quotient = " + num/den);</pre>

使用 if 和 if-else 语句的典型示例

一元二次方程的错误检查	<pre>double discriminant = b*b - 4.0*c; if (discriminant< 0.0) { System.out.println("No real roots"); } else { System.out.println((-b + Math.sqrt(discriminant))/2.0); System.out.println((-b - Math.sqrt(discriminant))/2.0); }</pre>
-------------	---

使用 if 和 if-else 语句的典型示例 (续)

52



While 循环 很多计算本质上是重复的，对于这类计算，Java 使用下面的基本结构进行处理：

```
while (<布尔表达式>) { <语句> }
```

while 语句与 if 语句具有相同的形式 (唯一的区别是使用关键字 while 而不是 if)，但是其含义却大不相同。while 语句向计算机发出的指令如下：如果布尔表达式为 false，则不执行任何操作；如果布尔表达式为 true，则执行语句序列 (就像使用 if 语句一样)，接着再次检查布尔表达式，如果表达式为 true，则再次执行语句序列，即只要表达式为 true，就不断地继续执行。我们将循环中的语句块称为循环的主体。与 if 语句一样，如果 while 循环体只有一个语句，则大括号是可以省略的。while 语句相当于一系列相同的 if 语句：

53

```
if (<布尔表达式>) { <语句> }
if (<布尔表达式>) { <语句> }
if (<布尔表达式>) { <语句> }
...
```

在某点下，其中一个语句中的代码会改变一些内容 (如布尔表达式中某个变量的值)，使布尔表达式为 false，则语句序列的执行被中断。

常见的 while 语句的编程用法中会维护一个整数值，以跟踪循环迭代的次数。我们从初始化好的某个值开始，然后每次循环时将值增加 1，在决定是否继续之前需要测试它是否超过预定的最大值。TenHellos（程序 1.3.2）就是使用 while 语句范例的一个简单例子，其中的关键语句是

```
i = i + 1;
```

作为一个数学方程式，这个语句是没有任何意义的，但是在 Java 赋值语句中它的意义非常清晰：先计算出 $i+1$ 的值，然后将计算出的结果赋值给变量 i 。如果计算前 i 变量的值是 4，那么计算后 i 的值是 5；如果 i 是 5，则变成 6，等等。在 TenHellos 程序中 i 的初始值是 4，循环体在执行结束前执行了 7 次，而 i 的值变成了 11。

程序1.3.2 第一个while循环语句

```
public class TenHellos
{
    public static void main(String[] args)
    { // 打印了十次Hello
        System.out.println("1st Hello");
        System.out.println("2nd Hello");
        System.out.println("3rd Hello");
        int i = 4;
        while (i <= 10)
        { // 打印第i次Hello
            System.out.println(i + "th Hello");
            i = i + 1;
        }
    }
}
```

该程序使用while循环来简单、重复地打印提示信息，运行效果如下所示。在第三行之后，每次要打印的信息差异仅是该行的索引值，因此我们设置一个变量 i 用于表示该索引。首先将 i 初始化为 4，然后进入一个 while 循环，我们在 `System.out.println()` 语句中使用 i 的值，并且在每个循环中增加它。打印第 10 个 Hello 后， i 的值为 11，循环结束。

% java TenHellos

```
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

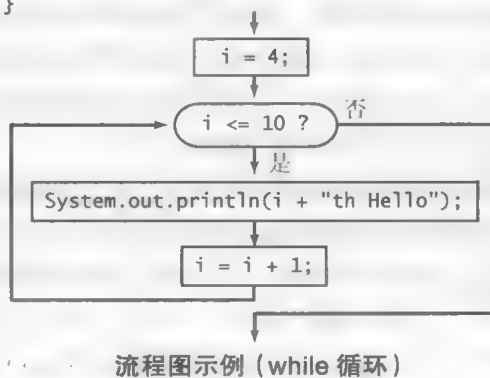
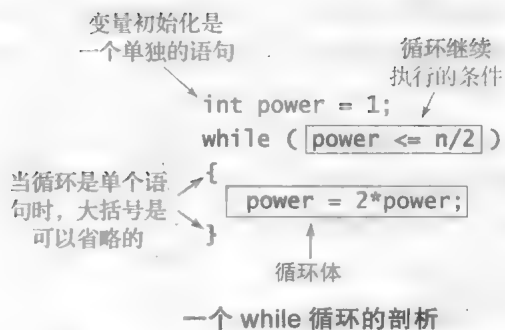
i	$i \leq 10$	output
4	true	4th Hello
5	true	5th Hello
6	true	6th Hello
7	true	7th Hello
8	true	8th Hello
9	true	9th Hello
10	true	10th Hello
11	false	

Trace of java TenHellos

对于这个简单的任务，使用 while 循环几乎没有什么价值，但是你很快就要解决一些复杂的任务，你需要将一些语句重复执行非常多次，以至于在没有循环的情况下无法完成这些任务。程序之间有无 while 语句有着很大的区别，因为 while 语句允许我们在程序中不限次数地执行特定的语句。借助 while 语句，我们可以在短程序中实现冗长的计算。这种能力打

开了编写程序的大门，而这些任务是在没有计算机的情况下无法解决的。但也有一个代价：随着程序越来越复杂，它们变得越来越难以理解。

```
int i = 4;
while (i <= 10)
{
    System.out.println(i + "th Hello");
    i = i + 1;
}
```



程序 PowersOfTwo (程序 1.3.3) 使用一个 while 循环打印出 2 的幂的表格。除了循环控制计数器 i 之外，它使用了一个变量 $power$ 来保存 2 的幂的值。循环体包含三个语句：一个用于打印 2 的当前幂，一个用于计算下一个幂 (将当前值乘以 2)，另一个用于递增循环控制计数器。

54
56

程序 1.3.3 计算 2 的幂

```
public class PowersOfTwo
{
    public static void main(String[] args)
    {
        // 打印 2 的前 n 次幂
        int n = Integer.parseInt(args[0]);
        int power = 1;
        int i = 0;
        while (i <= n)
        {
            // 打印第 i 个 2 的幂
            System.out.println(i + " " + power);
            power = 2 * power;
            i = i + 1;
        }
    }
}
```

n	循环终止的变量
i	循环控制计数器
$power$	当前循环计算的 2 的幂

该程序需要输入整型命令行参数 n ，并打印一个小于或等于 2^n 的 2 的幂的表格。每一次循环中都会增加变量 i 的值，同时将 $power$ 变量增加两倍。我们仅展示表格中前三行和最后三行，实际上程序会打印 $n+1$ 行。

```
% java PowersOfTwo 5
0 1
1 2
2 4
3 8
4 16
5 32
```

```
% java PowersOfTwo 29
0 1
1 2
2 4
...
27 134217728
28 268435456
29 536870912
```

在计算机中有许多情况需要用到 2 的幂方。你需要熟练地记住表格中的前 10 个值，也需要

注意一些有意思的数据，如 2^{10} 在 1000 (K) 左右， 2^{20} 大约是 100 万 (M)， 2^{30} 大约是十亿 (B)。

以 PowersOfTwo 为原型，还可以做出很多其他的计算模型，通过改变累积值的计算方法和循环控制量递增的方式，我们可以打印出很多种函数的取值表（详见练习 1.3.12）。

为了检查程序运行中的行为，需要仔细分析它的执行轨迹，这种技术叫作程序跟踪，是非常有意义的。例如跟踪 PowersOfTwo 程序，就是在每次循环前输出每个变量的值以及控制循环的布尔表达式的值。跟踪循环的操作可能非常烦琐，而且会输出大量的信息，但是跟踪通常是很有价值的，因为它可以清晰地展示程序每一步的操作。

PowersOfTwo 基本上是一个自我跟踪的程序，因为它通过循环来打印它的变量的值。显然，你可以在程序中通过 `System.out.println()` 语句来跟踪程序。现代编程环境提供了更加复杂的跟踪工具，但是使用输出语句（Print）这种方法更加简单有效，也非常可靠。你可以在循环程序中都加入 `print` 语句，以保证它们都精准地执行了你期望的操作。

在 PowersOfTwo 程序中有一个隐藏的陷阱，因为 Java 中最大的 `int` 数据是 $2^{31}-1$ ，但程序并没有检测到达这种边界值的情况，如果运行 `java PowersOfTwo 31`，你可能会发现最后一行的打印输出与你想象的不同：

```
...
1073741824
-2147483648
```

基于 Java 表示整数的方式，变量 `power` 变得太大会呈现负值。在 Java 中可以用的 `int` 型最大值是 `Integer.MAX_VALUE`，你可以用它来检测数据是否发生溢出。例如，可以改进程序 1.3.3，当用户输入的值过大时，可以输出一个错误消息。实际上，要想让你的程序对于所有可能的输入都产生正常工作结果，需要做的工作要远比想象的复杂。（对于类似的挑战，请参阅练习 1.3.16。）

举一个复杂一点的例子，假设你想要计算出小于或等于给定正整数 `n` 的最大的 2 的幂值，即如果 `n` 是 13，我们想要的结果就是 8；如果 `n` 是 1000，那么我们想要的结果是 512；如果 `n` 是 64，我们想要的结果是 64，等等。该计算可以使用 `while` 循环来实现：

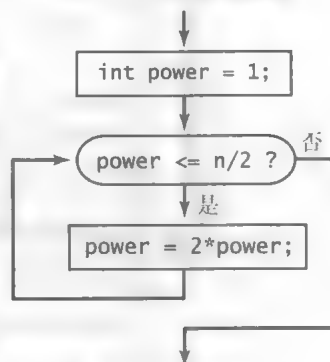
```
int power = 1;
while (power <= n/2)
    power = 2*power;
```

你需要花一些时间思考才能说服自己，这段简单的代码真的产生了期望的结果，可以通过观察来做到这一点：

- `power` 变量的值永远是 2 的幂。
- `power` 变量的值不可能比 `n` 大。
- `power` 变量的值在每次循环中都会增加，所以循环一定会终止。

i	power	i <= n
0	1	true
1	2	true
2	4	true
3	8	true
4	16	true
5	32	true
6	64	true
7	128	true
8	256	true
9	512	true
10	1024	true
11	2048	true
12	4096	true
13	8192	true
14	16384	true
15	32768	true
16	65536	true
17	131072	true
18	262144	true
19	524288	true
20	1048576	true
21	2097152	true
22	4194304	true
23	8388608	true
24	16777216	true
25	33554432	true
26	67108864	true
27	134217728	true
28	268435456	true
29	536870912	true
30	1073741824	false

运行命令 “`java PowersOfTwo 29`” 的跟踪结果



语句的流程图

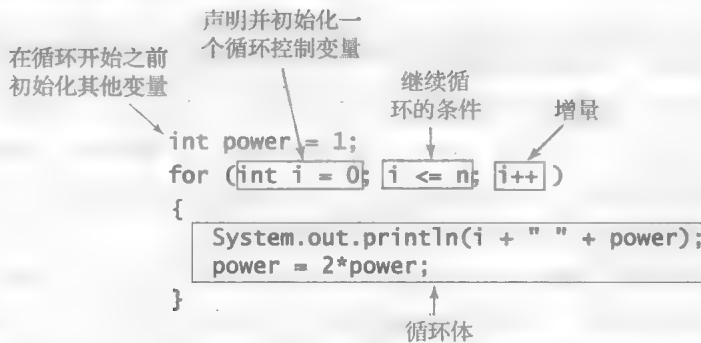
```
int power = 1;
while (power <= n/2)
    power = 2*power;
```

- 在循环终止后， $2 * \text{power}$ 的值会大于 n 。

推理这样的程序是如何工作的，对于理解 while 循环的工作原理往往非常重要。即便你编写的很多循环可能比这个简单，你也需要确保在每次循环过程中你所写的代码能够按照期望运行。

循环运行的次数可多可少，如程序 TenHellos 中只运行了几次，在 PowersOfTwo 中运行几十次，我们即将探讨的几个例子可能要运行几百万次。无论循环迭代多少次，以及循环体的大小，这些语句背后的逻辑是相同的。当我们写循环时，至关重要的是要理解每次循环中变量的变化。在编程的初期，你可以令循环语句运行少量的迭代次数，并添加输出语句来跟踪这些变量的值以检查你的理解是正确的。当你熟练以后，就可以彻底丢弃这些初学者的辅助工具，写出复杂的循环，从而真正释放计算机的能力。

for 循环 在后面的学习中我们会看到，利用 while 循环能够编写出各种应用程序。在继续学习更多例子前，我们来学习另外一个 Java 结构，以通过更加灵活的方式编写循环。这个结构和基本的 while 循环在原理上没有什么不同，但是使用得更加广泛，因为这种方式相比只用 while 语句书写的代码更加紧凑，可读性更好。



for 循环的剖析（用于打印 2 的幂）

for 语句。很多循环遵从这种形式：将一个索引变量初始化为某个值，接着使用 while 循环测试循环条件是否成立，循环条件往往是关于索引变量的判断，while 循环的循环体中最后一个语句用于递增索引变量。你也可以直接用 Java 中的 for 语句来表示这样的循环：

```
for (<初始化>; <布尔表达式>; <递增>)
{
    <循环体语句>
}
```

这段代码在绝大多数情况下都可以等价于下面的代码：

```
<初始化>;
while (<布尔表达式>)
{
    <循环体语句>
    <递增>;
}
```

你的 Java 编译器甚至可以为这两种循环的写法产生相同的执行代码。实际上，<初始化> 和 <递增> 部分可以是更复杂的语句，但我们几乎总是使用 for 循环来实现这种典型的“初始化-增量”的编程模式。例如，以下两行代码等同于 TenHellos（程序 1.3.2）中相应的代码行：

```
for (int i = 4; i <= 10; i = i + 1)
    System.out.println(i + "th Hello");
```

通常情况下，我们还会把这些代码写得更加简洁，这需要用到一些快捷的表示方法，我们下面来学习它们。

复合赋值语句。修改一个变量的值是编程中常见的事情，在 Java 中提供了各种各样的简写符号以实现这些功能。例如，以下四种语句可以实现将 i 的值增加 1：

```
i = i+1;   i++;   ++i;   i += 1;
```

类似地， $i--$ 、 $--i$ 或者 $i-=1$ 、 $i=i-1$ 实现将 i 的值减 1。大部分程序员在 for 循环中使用 $i++$ 或者 $i--$ 。 $++$ 或者 $--$ 结构通常只能应用于整数，而复合赋值（compound assignment）结构可以用于任何基本数字类型的算术运算符操作。例如，我们用 $\text{power} *= 2$ 或者 $\text{power} += \text{power}$ 来代替 $\text{power} = 2 * \text{power}$ 。所有这些语法都只是为了符号方便而提供的，没有什么特殊的意义。这些快捷的代码书写方式在 20 世纪 70 年代被 C 编程语言广泛使用，并已成为标准。它们经历了时间的考验，因为它们是最紧凑、优雅、易于理解的方案。当你学会了编写（和阅读）与这些符号相关的程序时，你会发现许多现代语言编程中都在使用这样的书写形式，而不仅仅是 Java。

作用域。变量的作用域（scope）是程序中能够通过名称来指向这个变量的代码区域。一般而言，变量的作用域是与变量声明在同一个块中的声明语句后面的语句。为此，for 语句的前导部分代码（即 for 语句中的初始化、布尔表达式、递增三个区域的代码——译者注）被认为与 for 循环体在同一个块中。因此，while 循环和 for 循环并不完全相同：在典型的 for 循环中，增量变量不能用于后面的语句中，而在对应的 while 循环中可能会用到。这种细微的区别通常是使用 while 循环而不是 for 循环的原因。

完成同一个计算任务，有多种不同的实现方式可以选择，至于选择哪一种，则是程序员的个人爱好问题，就如同作家在同义词中进行挑选，或者是决定句子使用主动语态还是被动语态的问题。关于如何编写一个程序，你不会找到简单高效的硬性规定，就像你不可能找到一个规则教你如何写好小说一样。你的目标应该是寻找适合你的风格——既能完成计算和编程任务，又可以被其他人所欣赏。

下面的表格中包含了几个在 Java 中常用的循环代码片段，其中一些代码与前面讲到的编程示例相关，还有一些是全新的代码，用来实现简单而直接的计算任务。为了巩固你对 Java 循环的理解，你可以自己找一些计算任务并用循环语句来解决它们，或者试着完成本节末的练习中的前面一部分。运行你自己编写的代码所获得的经验是其他方式不可替代的，你必须了解如何编写 Java 的循环代码。

计算小于或等于 n 的最大的 2 的幂	<pre>int power = 1; while (power <= n/2) power = 2*power; System.out.println(power);</pre>
计算有限个整数的和 ($1+2+\cdots+n$)	<pre>int sum = 0; for (int i = 1; i <= n; i++) sum += i; System.out.println(sum);</pre>
计算有限个整数的积（阶乘） ($n!=1\times 2\times \cdots \times n$)	<pre>int product = 1; for (int i = 1; i <= n; i++) product *= i; System.out.println(product);</pre>

打印一个函数值表	<pre>for (int i = 0; i <= n; i++) System.out.println(i + " " + 2*Math.PI*i/n);</pre>
计算标尺函数 (见程序 1.2.1)	<pre>String ruler = "1"; for (int i = 2; i <= n; i++) ruler = ruler + " " + i + " " + ruler; System.out.println(ruler);</pre>

使用 while 和 for 循环的典型示例 (续)

61

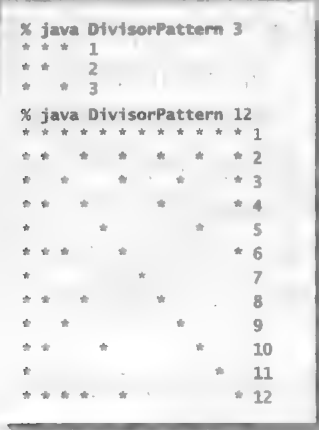
嵌套 if、while 和 for 语句与 Java 中的赋值语句或者任何其他语句在编程时地位是对等的，也就是说这些语句都可以出现在任何需要的地方，对于位置没有特殊的要求。特别需要说明的是，我们可以把它们中的一句或者多句用在另一个语句块中，当作语句的一部分，从而形成复合语句 (compound statement)。我们来看一个例子，程序 DivisorPattern (程序 1.3.4) 有一个 for 循环，而 for 循环的循环体里面还包含一个 for 循环 (嵌套的 for 循环体里包含的是一个 if-else 语句) 和一个 print 语句。这个程序用于打印一个由星号组成的图案，在第 i 行中，如果列号能够整除 i 或者被 i 整除，那么这个位置会有一个星号 (这个规律同样适用于列)。

程序1.3.4 你的第一个嵌套循环程序

```
public class DivisorPattern
{
    public static void main(String[] args)
    { // 输出一个正方形以表示整除的数字
        int n = Integer.parseInt(args[0]);
        for (int i = 1; i <= n; i++)
        { // 输出第i行
            for (int j = 1; j <= n; j++)
            { // 输出第i行的第j列元素
                if ((i % j == 0) || (j % i == 0))
                    System.out.print("* ");
                else
                    System.out.print(" ");
            }
            System.out.println(i);
        }
    }
}
```

n 行和列的数目
 i 行的索引值
 j 列的索引值

这一个程序要求输入命令参数 n ，然后用嵌套循环打印一个 $n \times n$ 的表格，对于表格中的第 i 行第 j 列，如果 i 除以 j 或者是 j 除以 i 余数是 0，就在第 i 行第 j 列上打印星号。

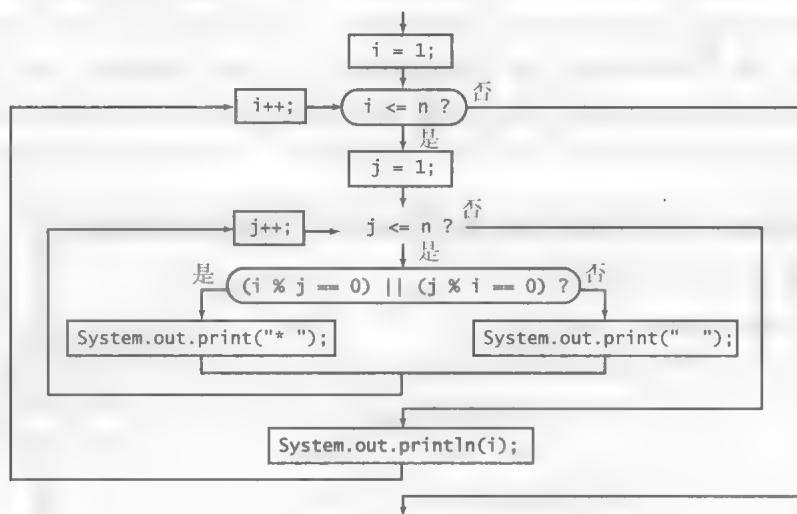


i	j	$i \% j$	$j \% i$	输出
1	1	0	0	*
1	2	1	0	*
1	3	1	0	*
2	1	0	1	*
2	2	0	0	*
2	3	2	1	
3	1	0	1	*
3	2	1	2	
3	3	0	0	*
				3

命令行参数输入3时
DivisorPattern和程序跟踪信息

为了使嵌套更加清晰，我们在程序代码中使用缩进。将 i 循环称为外部循环，将 j 循环称为内部循环。外部循环每迭代一次，内部循环都需要执行一整次循环（即把内部循环的循环控制变量从初始化递增到循环结束——译者注）。通常，了解这样一个新的编程结构的最好方法是研究程序的跟踪信息以明确每一步的迭代过程。

DivisorPattern 的控制流程更加复杂，你可以从它的控制流程图中看出来。通过这个图我们可以清楚地认识到，使用数量有限的简单控制流程结构在编程过程当中是至关重要的。使用嵌套的方法，即便程序的流程图很复杂，你也可以构建出非常容易理解的循环结构和判断语句。只需要一个或者两个嵌套就可以完成很多有用的计算，你将会看到，书中的许多程序都与 DivisorPattern 具有相同的结构。



DivisorPattern 的控制流程图

关于嵌套的第二例子，我们来看以下程序片段，这是一段用于计算所得税率的程序：

```

if      (income < 0) rate = 0.00;
else if (income < 8925) rate = 0.10;
else if (income < 36250) rate = 0.15;
else if (income < 87850) rate = 0.23;
else if (income < 183250) rate = 0.28;
else if (income < 398350) rate = 0.33;
else if (income < 400000) rate = 0.35;
else      rate = 0.396;
  
```

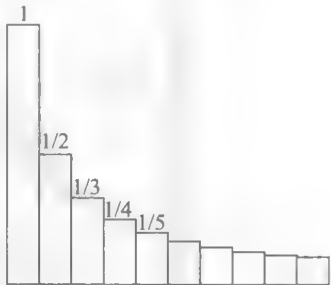
在这个例子中，多个 if 语句嵌套在一起，实现在多个互斥条件中进行判断。这种结构是一种特殊的写法，但是我们经常使用。通常情况下，在嵌套 if 语句时最好使用花括号来消除歧义。关于这个问题我们会在问答环节继续讨论，在练习中也会有更多的例子。

应用 一旦掌握了使用循环编写程序的能力，你就立刻打开了整个计算世界的大门。为了证明这一点，接下来的例子中我们会分析来自不同领域的计算任务，并用编程来解决问题。这些例子只用到了我们在 1.2 节中讲到的数据类型，但请放心，相同的机制可以为任何计算应用提供良好的效果。这些示例程序是经过精心研究的，通过研究它们，你可以更好地写出自己的循环程序。

我们在这里涉及的例子都是包含了一定运算量的。这几个例子与过去几个世纪数学家

和科学家面临的问题有关。虽然计算机已经存在了 70 多年，但是我们程序中使用的很多计算方法还是基于古老的传统数学方法，有些方法甚至可以追溯到古代。

计算有限数和。PowersOfTwo 程序是一个很好的程序模板，你在编程中可能会经常使用到。它使用两个变量，一个用作控制循环的索引值，另一个用于累加计算结果。程序 HarmonicNumber (程序 1.3.5，即用于计算谐波数——译者注)使用的就是这个模板，以计算有限个数字的和 $H_n=1+1/2+1/3+\cdots+1/n$ 。这些数字通常称为谐波数 (harmonic number)，在离散数学中会经常用到。谐波数是对数函数的离散模拟，也可以用于近似计算 $y=1/x$ 曲线下的面积。你也可以使用程序 1.3.5 作为模板来计算其他有限个数字的和 (见练习 1.3.18)。



64

程序1.3.5 谐波数 (一)

```
public class HarmonicNumber
{
    public static void main(String[] args)
    { // 计算第n个谐波数
      int n = Integer.parseInt(args[0]);
      double sum = 0.0;
      for (int i = 1; i <= n; i++)
      { // 将第i项添加到总和中
        sum += 1.0/i;
      }
      System.out.println(sum);
    }
}
```

n

i

sum

用于求和的数字的个数

循环索引

累加结果

这个程序要求输入一个int型的命令行参数n，然后计算第n个谐波数的值。从数学角度分析，对于n值较大的情况，谐波数的值大约是 $\ln(n)+0.577\ 21$ 。注意， $\ln(1\ 000\ 000)+0.577\ 21\approx14.392\ 72$ 。

% java HarmonicNumber 2
1.5

% java HarmonicNumber 10
2.9289682539682538

% java HarmonicNumber 10000
7.485470860550343

% java HarmonicNumber 1000000
14.392726722864989

计算平方根。Java 中的 Math 库 (数学库) 是如何实现的? 我们来研究平方根函数 Math.sqrt()。程序 Sqrt (程序 1.3.6) 展示了一种实现方法。这段程序的计算方法使用的是 4000 年前巴比伦人所提出的迭代计算方法。这种算法也是 17 世纪由 Isaac Newton 和 Joseph Raphson 开发的通用求平方根方法的特殊形式，因此也称为牛顿法。在一般情况下，牛顿法可以用于求解给定函数 $f(x)$ 的根，即 $f(x)$ 值为 0 时， x 的值。对于某个估计值 t_i ，通过在点 $(t_i, f(t_i))$ 处绘制与曲线 $y=f(x)$ 相切的线，并将该切线与 x 轴的交点记为 t_{i+1} ， t_{i+1} 就是新的估计值。从最初的估计值 t_0 开始，不断迭代这个过程，就会越来越接近函数的根。

65

程序1.3.6 牛顿法

```

public class Sqrt
{
    public static void main(String[] args)
    {
        double c = Double.parseDouble(args[0]);
        double EPSILON = 1e-15;
        double t = c;
        while (Math.abs(t - c/t) > EPSILON * t)
        { // 用t和c/t的平均值来代替t
            t = (c/t + t) / 2.0;
        }
        System.out.println(t);
    }
}

```

c	参数
EPSILON	误差容忍度
t	c的平方根的估计

该程序使用正的浮点数 c 作为命令行参数，并使用牛顿法计算 c 的平方根，精度计算到小数点后15位。

```

% java Sqrt 2.0
1.414213562373095
% java Sqrt 2544545
1595.1630010754388

```

迭代	t	c/t
	2.0000000000000000	1.0
1	1.5000000000000000	1.3333333333333333
2	1.4166666666666665	1.4117647058823530
3	1.4142156862745097	1.4142114384748700
4	1.4142135623746899	1.4142135623715002
5	1.4142135623730950	1.4142135623730951

java Sqrt 2.0 的跟踪信息

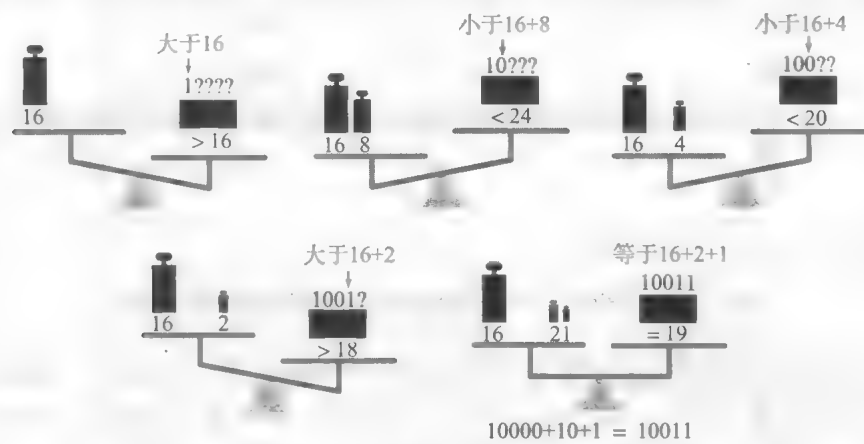
66

计算一个正数 c 的平方根等同于找到函数 $f(x)=x^2-c$ 的一个正根。程序 Sqrt 就是牛顿法在这种特殊情形下的代码实现（完整的实现见练习 1.3.19）。从估值 $t=c$ 开始，如果 t 等于 c/t ，那么 t 就等于 c 的一个平方根，这样计算就结束了。如果不相等，那么用 t 和 c/t 之间的平均值来替代我们之前的估值 t 。使用牛顿法，我们求得一个值的平方根仅仅只需要5次循环迭代，结果可以精确到小数点后的15位。

牛顿法在科学计算当中非常重要，因为求根问题是广泛存在的，而类似的迭代方法在这些问题中也是有效的。甚至许多还没有从分析的角度找到求解方法的问题也可以用牛顿法求解。同样，在 Java 的 Math 库中没有提供支持的数学函数也可以这样求解。有了这个方法和计算机的计算能力，我们认为可以找到任何我们需要的函数的值；而在计算机出现之前，科学家和工程师不得不使用查表法或手动计算。为了使手动计算尽可能高效，人们提出了多种计算技术和方法，当我们使用计算机时，这些技术和方法可以更加高效地发挥作用。牛顿法就是一个典型的例子。计算数学函数值的另一个有用的方法是使用泰勒级数展开（参见练习 1.3.37 和练习 1.3.38）。

数制转换。程序 Binary（程序 1.3.7）能够将命令行参数输入的十进制数字转换为二进制（基数为2）并打印出来。它的方法是将一个数分解为若干个2的幂的和。例如，19的二进制表示为10011，也就意味着 $19=16+2+1$ （ $19=2^4+2^1+2^0$ ）。为了计算 n 的二进制表示，我们需要计算出所有小于或等于 n 的2的幂值，将它们按降序排列，用以确定哪些可以用于二进制分解（被采用的值，对应二进制表示中相应的位为1）。这个过程相当于使用天平来称量一个物体，使用重量是2的幂的砝码，首先我们找到不超过物体的最大的砝码。然后，按照递减的顺序考虑砝码，我们每添加一个砝码即测试是否超过物体的重量，如果是，则我们

拿掉砝码，否则我们留下砝码，并尝试下一个砝码。每一个砝码对应的是对象重量的二进制表示中的一位。如果移除砝码，那么对应物体重量的二进制位的表示是 0。



利用天平模拟二进制转换

程序1.3.7 二进制转换

```
public class Binary
{
    public static void main(String[] args)
    {
        // 打印n的二进制表示
        int n = Integer.parseInt(args[0]);
        int power = 1;
        while (power <= n/2)
            power = 2*power;
        // 现在的幂值是小于或等于n的最大的2的幂值
        while (power > 0)
        {
            // 降序排列2的幂
            if (n < power) { System.out.print(0); }
            else { System.out.print(1); n -= power; }
            power /= 2;
        }
        System.out.println();
    }
}
```

该程序要求命令行参数输入正整数n，然后打印n的二进制表示，使用的就是书中讲到的降序排列2的幂的方法。

```
% java Binary 19
10011
% java Binary 100000000
101111101011110000100000000
```

在 Binary 程序中，变量 power 代表的是当前已经尝试过称重的砝码，而变量 n 代表的是超出砝码重量部分的物体重量（为了模拟平衡超出部分，我们从变量 n 中减去已测试过的砝码重量）。变量 power 的值按照 2 的幂次递减。若 power 的值超过 n，Binary 程序打印输出 0；否则，打印 1 并从 n 中减去当前 power 的值。和往常一样，我们展示了每次循环迭代的轨迹（n 的值，power 的值，n<power 的判断结果，每次迭代输出的一位二进制数），以帮助理解程序的执行过程。跟踪信息表的最右边从上到下看，可以看到输出结果为 10011，即

19 的二进制形式。

n	二进制表示	power	power>0	二进制表示	n<power	输出
19	10011	16	是	10000	否	1
3	0011	8	是	1000	是	0
3	011	4	是	100	是	0
3	01	2	是	10	否	1
1	1	1	是	1	否	1
0		0	否			

运行“java Binary 19”(转换过程和计算过程的跟踪信息)

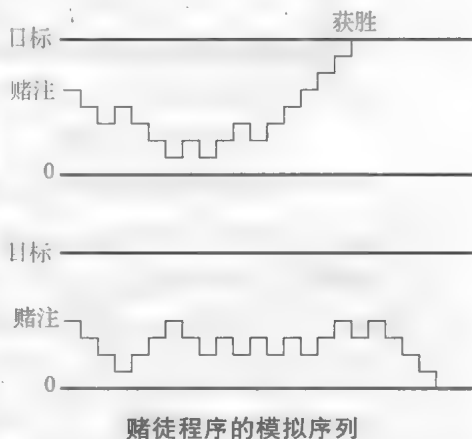
编写计算机程序时常见的一类任务是将数据从一种表示形式转换到另一种表示形式。转换思想强调的是信息(例如一天中的小时数)和信息具体表示形式(24 或 11000)之间的区别。值得一提的是,计算机内部是采用二进制数的形式来表示 int 变量的。

模拟。下一个例子与我们前面一直在探讨的例子在性质上有所不同,但它代表了一种常见的情形。我们常常使用计算机模拟现实世界中可能发生的情况,以便做出正确的选择。我们现在要探讨的这个例子来自于著名的赌徒破产原则。这类问题已经得到较为深入的研究,结论是,一开始给赌徒一定的财产,假设参与的是一个公平的赌局,每次下注 1 美元,赌徒最终总会破产。但是,当我们在游戏中设置一些限制,各种问题随之产生。例如,假设赌徒在达成某一目的之后决定提前离开,赌徒有没有机会赢?要最终赢得胜利,需要多少赌资?赌徒在过程中可获得的最高金额是多少?

程序 Gambler(程序 1.3.8)是一个模拟程序,可以帮助回答以上这些问题。在程序中,我们做了一系列的尝试,每次用 Math.random() 来模拟赌徒下的赌注,一直持续到赌徒破产或达成目标,跟踪记录赌徒到达目标所需的赌博次数和投注次数。我们将实验程序运行若干次,将得到的结果取平均值并打印出来。你可能希望运行这个程序并通过命令行参数来设置一些必要的值,这不一定是用于规划你下一次的赌场之旅,而是为了帮助你思考下面的问题:模拟是否能够准确地反映现实生活中会发生什么?需要进行多少次试验才能得到准确的答案?进行这种模拟的计算限制是什么?模拟广泛应用于经济、科学和工程中,以及回答在任何模拟中都重要的类似问题。

在 Gambler 这个例子当中,我们验证了概率论的经典结果:赌徒成功的概率是赌注与目标的比率,预期的投注次数是赌注和期望收益的乘积(收益是指目标和赌注之间的差)。例如,如果你去蒙特卡洛尝试把 500 美元变成 2500 美元,那么你有一个合理的(20%)成功概率,但你需要在每次赌 1 美元的赌局上赌 100 万次。如果你尝试将 1 美元变成 1000 美元,则只有 0.1% 的成功概率,并且需要赌大约 999 次(最有可能的还是破产)。

模拟和分析是相辅相成的,一方验证另一方。实际上,模拟的意义在于它可以求解一些难以用分析



解决的问题，例如，假设赌徒可能意识到没有足够多的时间进行 100 万次投注，因此可以提前设定投注次数的上限。在这个例子中，赌徒能赚取多少钱回家？你可以通过对程序 1.3.8（见练习 1.3.26）进行简单修改来解决这个问题，但是通过数学分析来解决这个问题并不容易。

程序1.3.8 赌徒破产模拟

```
public class Gambler
{
    public static void main(String[] args)
    { // 开始运行实验时资产为stake
      // 赌博活动终于资产为0或者资产为goal
      int stake = Integer.parseInt(args[0]);
      int goal = Integer.parseInt(args[1]);
      int trials = Integer.parseInt(args[2]);
      int bets = 0;
      int wins = 0;
      for (int t = 0; t < trials; t++)
      { // 运行实验
        int cash = stake;
        while (cash > 0 && cash < goal)
        { // 模拟一次赌博
          bets++;
          if (Math.random() < 0.5) cash++;
          else cash--;
        } // 现金是0（破产）或者是goal（获胜）
        if (cash == goal) wins++;
      }
      System.out.println(100*wins/trials + "% wins");
      System.out.println("Avg # bets: " + bets/trials);
    }
}
```

stake	初始赌资
goal	设定的离开赌场的目标
trials	试验次数
bets	下注数
wins	获胜次数
cash	手上的现金

本程序需要三个命令行参数，即三个int类型变量，stake、goal和trials。分别表示赌注、目标和试验次数。程序中的内层循环模拟一个拥有赌资stake的赌徒做了一系列1美元的赌注，持续到破产或者达到资产为goal。该程序的运行时间与平均下注试验次数trials成正比。例如，下面的第三个命令会产生将近1亿次的随机实验。

```
% java Gambler 10 20 1000
50% wins
Avg # bets: 100
% java Gambler 10 20 1000
51% wins
Avg # bets: 98
```

```
% java Gambler 50 250 100
19% wins
Avg # bets: 11050
% java Gambler 500 2500 100
21% wins
Avg # bets: 998071
```

整数分解。素数是指大于 1 并且只能被 1 和它本身整除的整数。整数分解（也称为素因子分解）是指将整数分解为若干个素数，这些素数的乘积是整数本身。例如， $3\,757\,208 = 2 \times 2 \times 2 \times 7 \times 13 \times 13 \times 397$ 。程序 Factors（程序 1.3.9）可以用于计算任何给定正整数的素因子分解。与我们已经见过的程序相比，对于以前的程序我们是可以用计算器甚至用纸和笔在几分钟内计算出来的，而这个程序的计算任务如果没有计算机是不可能完成的。如何尝试找到类似 287994837222311 这样数字的素数因子呢？你可能能够马上发现因子 17，但是即便使用了计算器你也要花费一段时间才能找到 1739347。

程序1.3.9 整数分解

```

public class Factors
{
    public static void main(String[] args)
    { // 打印n的素数分解
        long n = Long.parseLong(args[0]);
        for (long factor = 2; factor <= n/factor; factor++)
        { // 测试factor是不是因子
            while (n % factor == 0)
            { // 从n中除掉factor, 并把factor打印出来
                n /= factor;
                System.out.print(factor + " ");
            } // 任何n的因子都必须大于factor
        }
        if (n > 1) System.out.print(n);
        System.out.println();
    }
}

```

n | 未被分解的部分
factor | 可能的分解因子

该程序要求从命令行输入正整数 n 作为参数, 并打印 n 的素因子分解结果。代码非常简单, 但是要理解它的原理需要花一些工夫 (见正文)

```
% java Factors 3757208
2 2 2 7 13 13 397
```

```
% java Factors 287994837222311
17 1739347 9739789
```

虽然 Factors 程序很短, 但你肯定会需要一些思考来说服自己: 它能够对任何给定的整数产生所需的结果。和往常一样, 我们采用跟踪的手段, 对于外层的 for 循环, 打印出每一次迭代开始的值。这对于我们理解计算过程是一个很好的方案。对于初始值 n 是 3757208 的情况, 当 factor 为 2 时, 内部的 while 循环会迭代 3 次, 即要从 n 中分解出 3 个值为 2 的因子, 此时 n 的值变成 469651; 当 factor 是 3、4、5 和 6 时则迭代 0 次, 因为这些数字无法整除 n , 以此类推。给程序输入几个不同的值, 跟踪执行的过程可以发现程序求解的基本过程。为了确保程序对于所有的输入都能按照预期执行, 我们会解释期望的每个循环做的是是什么。while 循环会将 n 中所有值为 factor 的因子都打印出来并将其从待分解数中移除。这个程序的关键在于, 每次 for 循环开始迭代前, n 当中肯定没有大于 2 并且小于 factor-1 的因子。因此, 如果 factor 不是素数, 它将不能整除 n ; 如果 factor 是一个素数, 那么 while 循环将会测试是否可以整除并将 factor 移除。一旦我们知道 n 没有小于或等于 factor 的因子, 我们也知道没有因子大于 n /factor, 所以当 factor 大于 n /factor 时, 我们也就不需要再继续循环了。

因子	数	输出	因子	数	输出
2	3757208	2 2 2	12	67093	
3	469651		13	67093	13 13
4	469651		14	397	
5	469651		15	397	
6	469651		16	397	
7	469651	7	17	397	
8	67093		18	397	
9	67093		19	397	
10	67093		20	397	
11	67093				

397

一个更简单而直接的实现方式是，我们可以使用语句 (`factor<n`) 来作为终止 `for` 循环的条件。即使考虑到现代计算机的计算速度，这样做也会对我们可以计算的数字的大小产生巨大的影响。在练习 1.3.28 中我们希望使用这种简单的实现方式，从而体会这个小小的修改给程序性能带来的变化。如果按照这样简单的实现方式，在每秒可以执行数十亿次的计算机上，我们可以在几秒钟内计算出 10^9 数量级的数字的素因子分解；而如果使用 (`factor<= n/factor`) 作为测试条件，我们可以在这个运行时间内完成对 10^{18} 数量级的数字的素因子分解。循环使我们有能力解决艰难的问题，但也会使我们的简单程序运行得非常慢，所以我们在编程的过程中必须时刻注意计算方法对性能的影响。

在现代密码学的应用中，在一些重要的情况下我们需要对非常巨大的数字（数百或数千位数字）做素因子分解，而这时即使使用计算机，完成这样的计算任务也是非常困难的。

其他形式的条件和循环结构 为了更全面地介绍 Java 语言，我们在这里再介绍四种控制结构，你不必考虑将这些结构用于你自己编写的每个程序，因为你遇到它们的频率远远低于 `if`、`while` 和 `for` 语句。在熟练使用 `if`、`while` 和 `for` 语句前不用担心如何使用这些结构。你可能会在书籍或者网络中看到它们，但是许多程序员并不使用它们，且本书在这一节之外也很少使用它们。

`break` 语句。在某些情况下，我们想立即退出循环，而不是让其运行直到循环结束。Java 为此提供了 `break` 语句。例如，以下代码能够检测给定整数 `n>1` 是否为素数：

```
int factor;
for (factor = 2; factor <= n/factor; factor++)
    if (n % factor == 0) break;
if (factor > n/factor)
    System.out.println(n + " is prime");
```

退出这个循环有两种不同的方法：`break` 语句被执行（因为 `n` 能够被 `factor` 整除，因此 `n` 并不是一个素数）或者循环条件不满足（因为对于满足 `factor<=n/factor` 的每种情况，`n` 都无法整除 `factor`，则 `n` 是素数）。注意，我们在 `for` 循环的外部声明 `factor`，而不是在 `for` 循环的初始语句中，这样 `factor` 的作用域就能拓展到循环以外了。

`continue` 语句。Java 还提供了一种跳转到下一个循环迭代的方法：`continue` 语句。当在 `for` 循环的主体内执行 `continue` 语句时，控制流将直接传递给递增语句以用于下一次循环迭代。

`switch` 语句。`if` 和 `if-else` 语句允许在控制流中产生一个或者两个可供选择的方向。有时候，一个计算会产生两种以上互斥的选择。我们可以使用一系列 `if-else` 语句（如本节前面讨论的税率计算问题），但是 Java 的 `switch` 语句提供了一个更加直接的解决方案。让我们列举一个典型的例子。如果我们想把用来表示星期几的一个 `int` 型变量用更加友好的方式显示出来（如练习 1.2.29 的解决方案），用 `switch` 语句更加容易，如下所示：

```
switch (day)
{
    case 0: System.out.println("Sun"); break;
    case 1: System.out.println("Mon"); break;
    case 2: System.out.println("Tue"); break;
    case 3: System.out.println("Wed"); break;
    case 4: System.out.println("Thu"); break;
    case 5: System.out.println("Fri"); break;
    case 6: System.out.println("Sat"); break;
}
```


当你有一个看起来很长并且很有规则的 if 语句时，你可以尝试使用 switch 语句，或者 1.4 节中描述的替代方法。

do-while 循环。另一个写循环的方法是使用下面的模板：

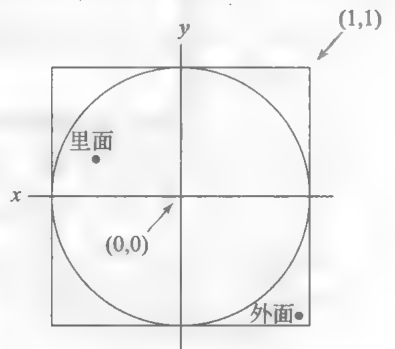
```
do { <语句> } while (<布尔表达式>);
```

这个语句的意思和以下语句基本一样：

```
while (<布尔表达式>) { <语句> }
```

只是布尔条件的第一个测试会被省略。如果布尔条件最初是成立的，那么二者没有区别。为了展示 do-while 的用途，考虑这样一个例子，如果要在单位圆中生成随机分布的点，我们可以使用 Math.random() 独立地生成 x 和 y 坐标，以获得随机分布在以原点为中心的 2×2 正方形中的点。大多数点同时也会落在圆内，所以我们只是拒绝那些没有落在圆内的。我们总是希望至少生成一个点，所以 do-while 循环对于这个计算来说是比较理想的。下面的代码设置 x 和 y ，使得点 (x, y) 随机分布在单位圆中：

```
do
{ // 将x和y缩放为 (-1, 1) 中的随机变量
  x = 2.0*Math.random() - 1.0;
  y = 2.0*Math.random() - 1.0;
} while (x*x + y*y > 1.0);
```



由于圆的面积是 π ，矩形的面积是 4，因此循环预期的迭代次数为 $4/\pi$ (大约是 1.27)。

75

无限循环（又称死循环）在使用循环编写程序前，你需要思考一下这个问题：如果一个 while 循环中的循环条件总是成立的，会发生什么情况？对于你迄今为止学习到的语句，都有可能出现类似这样的意想不到的情况，但是无论发生哪种情况你都要学会去处理。

第一种情况：我们假设这样的无限循环调用了 System.out.println() 函数。例如，如果 TenHellos 程序中循环继续的条件为 $(i > 3)$ 而不是 $(i \leq 10)$ ，那么这个条件将会总为真，接下来会发生什么呢？在当前的开发环境中，我们调用 print 时，它的作用是在显示器的终端窗口中显示相应内容。在终端窗口中尝试显示无限多的行，具体会产生什么样的结果是由操作系统的设计者决定的。但如果 print 的作用是在纸上打印字符，那么就有可能用完所有的打印纸或者需要拔掉打印机电源才能使它停下来。在终端窗口中，我们也需要一个停止打印的操作。在运行自己的循环程序前，你需要知道如何在 System.out.println() 调用的无限循环中“拔掉开关”，然后再修改上述 TenHellos 程序并测试它的运行结果。在大多数程序中，<Ctrl-C> 的意思是停止当前程序，在这里可以用来停止循环打印。

第二种情况：无限循环的程序运行之后可能什么都没发生。如果你的程序中有一个无限循环并且没有产生任何输出，那么程序会被锁在这个循环当中而你也将看不到任何结果。当处于这种情况时，你可能要检查循环以确保它一定会触发退出条件，但是问题常常是无法确定哪个循环在什么情况下出了问题。在查找问题的过程中，你可以通过插入 System.out.println() 调用函数去跟踪循环的执行。如果加入的这些函数正好处于一个无限循环中，那么

我们就回到了上一段中讨论的问题，你可以想办法停止窗口中的打印信息，并且这些输出信息会告诉你无限循环发生在什么地方。

你可能无法确定（或者可能也并不在意）一个循环是无限的还是只是很长。由于 int 数溢出的原因，即使是 BadHellos 程序最终也会在打印超过十亿行以后终止。如果你调用程序 1.3.8 时使用如下参数：java Gambler 100000 20000 100，你可能永远也无法等到答案。后面你将会学到估算程序运行时间的方法。

为什么 Java 没有直接检测无限循环并警告我们呢？答案可能会让你很惊讶。一般来说，我们是无法检测到无限循环的。虽然这个结论有违常理，但却是理论计算机科学的基本结果之一。

总结 作为参考，附表列出了我们在本节中探讨过的程序。它们是我们可以用 if、while 和 for 语句及内置数据类型来编写短程序以解决的典型任务。完成这些计算的编程，也是熟悉基本的 Java 控制流程结构语句的一种有效方法。

```
public class BadHellos
{
    int i = 4;
    while (i > 3)
    {
        System.out.println
            (i + "th Hello");
        i = i + 1;
    }
}
```



```
% java BadHellos
1st Hello
2nd Hello
3rd Hello
5th Hello
6th Hello
7th Hello
...
```

76

一个无限循环的例子

程序	描述
Flip	模拟硬币的投掷
TenHellos	第一个循环语句
PowersOfTwo	计算并打印一个函数的取值表
DivisorPattern	你的第一个嵌套循环
Harmonic	计算有限个数字的和
Sqrt	经典的迭代算法
Binary	基本的数制转换
Gambler	利用嵌套循环进行模拟
Factors	for 循环嵌套 while 循环

本节中程序的总结

为了学习如何使用条件语句和循环语句，你需要练习编写并调试 if、while 和 for 语句。本节结尾的练习为你提供了这样的机会。对于每个练习，你需要编写一个 Java 程序，运行并测试它。所有的程序员都知道，第一个程序运行很难按照计划来执行，所以你需要了解你的程序，并明白它的每一步应该做什么。首先，使用跟踪的方法来检查你的理解是否正确。当你逐渐熟悉之后，你就可以在编写循环程序的同时想象跟踪语句的输出结果。在编程的过程中，你需要问自己以下几个问题：在第一次循环迭代后，变量的值是什么？第二次呢？最后一次呢？在什么情况下这个程序会陷入无限循环？

循环和条件语句提升了我们的计算能力：if、while、for 语句使我们从仅能编写简单的直线程序转换为可以编写任意复杂的控制流程程序。在接下来的章节中，我们还会有巨大的收获，我们将学习如何处理大数据量的输入，并学习如何定义和处理除简单数值类型以外的数据类型。本节中的 if、while 和 for 语句在我们接下来的程序中将扮演至关重要的角色。

77

问答环节

问：= 和 == 之间的区别是什么？

答：我们在这里重复这个问题是为了提醒你，当你想对两个数字是否相等进行布尔判断时不要使用“=”，而一定要使用“==”。表达式 (x=y) 是将 y 的值赋值给 x，而表达式 (x==y) 是测试两个变量是否有相同的值。在一些编程语言中，这种差异可能会对程序造成严重的破坏，并且难以发现，但是 Java 的类型安全通常能及时发现这种问题。例如，如果我们在程序 1.3.8 中输入 (cash=goal) 而不是 (cash==goal)，编译器将自动为我们发现这个问题：

```
javac Gambler.java
Gambler.java:18: incompatible types
found   : int
required: boolean
if (cash = goal) wins++;
      ^
1 error
```

cash 和 goal 都不是布尔类型，因此编译器可以发现这个错误。但是，当 x 和 y 都是布尔值变量时就需要格外小心，如果你写成了 if (x=y)，这将被视为一个赋值语句，它的作用是将 y 的值分配给 x，它的计算结果就是 y 的值。为了避免这类错误，你应该把代码写成 if(!isPrime)，而不是 if(isPrime = false)。

问：所以需要在写循环和条件语句时小心注意，应该使用“==”而不是“=”。还有什么值得我特别注意的吗？

答：另一个常见的错误是在多个组合的条件语句块或者循环语句块中忘记大括号，例如，下面这段代码是 Gambler 程序的另一种实现方式：

```
for (int t = 0; t < trials; t++)
    for (cash = stake; cash > 0 && cash < goal; bets++)
        if (Math.random() < 0.5) cash++;
        else cash--;
    if (cash == goal) wins++;
```

代码看起来正确，但实际上并不正确。因为第二个 if 在 for 循环之外，所以只会执行一次。基于许多程序员的编程习惯，无论循环体或者条件语句包含几条语句，都要使用大括号将它们包括起来，以界定一个循环和条件的边界。这是一个很好的编程习惯，能够有效避免这类错误。

问：还有其他的吗？

答：第三个经典陷阱是嵌套使用 if 语句时会产生歧义：

```
if <表达式1> if <表达式2> <语句 A> else <语句 B>
```

在 Java 中这等同于：

```
if <表达式1> { if <表达式2> <语句 A> else <语句 B> }
```

你可能会觉得它应该按照下面的模式执行：

```
if <表达式1> { if <表达式2> <语句 A> } else <语句 B>
```

再次提醒，明确使用括号来划定语句体是避免这种错误的有效方法。

问：是否存在某些情况必须使用 for 循环而不能使用 while 循环？或者反过来。

答：没有，一般来说，当你的任务可以划分成初始化变量、变量增量和循环条件测试三部分，并且你不需要在循环外再使用循环控制变量时，应该使用 for 循环。但是也可以使用 while 循环实现完全等价的功能。

问：循环控制变量声明的位置有什么规则吗？

答：对于这个问题大家有不同的见解。在一些早期的编程语言中，要求所有变量声明放在语句块开始之前，很多程序员都习惯了这种方式，并且很多代码也是遵循这个约定书写的。但是，在第一次使用变量时再声明它也是非常合理的一种方法，特别是对于 for 循环，在一般情况下，循环变量在 for 循环之外也很少被用到。从另一方面说，在循环外部需要用到循环控制变量的情况也并不少见，如我们在讲解 break 时的示例代码中做素数检测那部分。

问：i++ 和 ++i 之间的区别？

答：作为一个独立的语句，这两者没有任何的区别。但如果作为一个部分出现在表达式中，两者还是有差异的。它们都可以使得 i 值加 1，但是 ++i 返回的是增加以后的值，而 i++ 返回增加之前的值。在本书中，我们尽量避免使用类似 x=++i 这样的代码，因为它除了赋值还会改变原来变量的值，这是不安全的。我们并不想花费过多的精力分析两者在 for 循环中的区别，因此我们在 for 循环中统一使用 i++，在独立的语句中也用 i++。只有在特殊情况下我们才会使用 ++i，那时我们会专门提醒注意并解释为什么要使用它。

79

问：在一个 for 循环中，< 初始化语句 > 和 < 增量语句 > 可以做很多复杂的事情，远不止声明变量、初始化和更新循环增量语句。我们该如何利用这个功能？

答：< 初始化语句 > 和 < 增量语句 > 可以通过逗号分隔的一系列语句。这种表示法使得我们除了可以初始化或者修改循环控制变量以外，还可以操作其他变量。在有些情形中，这种写法可以导致代码十分紧凑。例如，以下两行代码可以替代 PowersOfTwo 程序（程序 1.3.3）中 main() 语句块的最后 8 行。

```
for (int i = 0, power = 1; i <= n; i++, power *= 2)
    System.out.println(i + " " + power);
```

这样压缩代码其实并没有什么必要，所以尽可能不要这么写，尤其是对于初学者而言。

问：在 for 循环中可以使用 double 类型的变量作为循环控制变量吗？

答：这是可以的，但通常不要这样做。考虑以下循环：

```
for (double x = 0.0; x <= 1.0; x += 0.1)
    System.out.println(x + " " + Math.sin(x));
```

这个循环会迭代多少次？迭代的次数取决于测试两个 double 类型的值是否相等，由于浮点数的精度问题可能永远不能获得你预期的结果。

问：在循环中还有什么棘手的问题吗？

答：for 循环语句中的各个组成块都是可以省略的，初始化语句、布尔表达式、增量语句和循环体每一个部分都可以是空语句（即不存在）。但是，为了便于理解和阅读，最好使用 while 循环来代替有空语句的 for 循环。在本书的代码中，我们将避免出现这样带有空语句的 for 循环。

```
int power = 1;
while (power <= n/2)
    power *= 2;           增量语句为空
for (int power = 1; power <= n/2; )
    power *= 2;
for (int power = 1; power <= n/2; power *= 2)
    ; ← 循环体为空
```

三个等价的循环

80

- 1.3.1 编写一个程序，需要输入三个整型命令行参数，如果三个数相等则在终端窗口中打印“equal”，否则打印“not equal”。
- 1.3.2 编写一个更通用、更健壮的 Quadratic 程序（程序 1.2.3），用于求解并打印多项式 ax^2+bx+c 的根。如果判别式是负数则打印合适的提示信息；如果 a 是 0，则给出合理的提示信息（避免除数为零的情况）
- 1.3.3 以下每个语句有什么问题（如果有的话）？
 - a. `if (a>b) then c=0;`
 - b. `if a>b { c=0; }`
 - c. `if (a>b) c=0;`
 - d. `if (a>b) c=0 else b=0;`
- 1.3.4 编写一个代码段，如果双精度变量 x 和 y 的值都严格在 0 和 1 之间则打印，否则为假。
- 1.3.5 编写一个 RollLoadedDie 程序，打印一个不公平的掷骰子游戏的结果，使得获得 1、2、3、4 或者 5 的概率是 $1/8$ ，获得 6 的概率是 $3/8$ 。
- 1.3.6 通过添加代码改进练习 1.2.25 来提高你的解决方案，检查命令行参数在公式中是否有效，并且添加代码来打印如果不是这种情况下的错误消息。
- 1.3.7 假设 i 和 j 都是 `int` 类型。执行以下每个语句， j 的值是多少？
 - a. `for (i=0, j=0; i<10; i++) j+=i;`
 - b. `for (i=0, j=1; i<10; i++) j+=j;`
 - c. `for (j=0; j<10; j++) j+=j;`
 - d. `for (i=0, j=0; i<10; i++) j+=j++;`
- 1.3.8 重写 TenHellos 来创建一个 Hellos 程序，该程序采用命令行参数来设置要打印的行数，你可以假设参数小于 1000。提示：使用 `i%10` 和 `i%100` 来确定打印第 i 个 Hello 应该使用 `st`、`nd`、`rd` 还是 `th`。
- 1.3.9 编写一个程序，使用一个 `for` 循环和一个 `if` 语句，打印 1000 到 2000 之间的整数，每行 5 个数字。提示：使用 `%` 操作。
- 1.3.10 编写一个程序，要求输入 `int` 型命令行参数 n ，然后使用 `Math.random()` 生成并打印 n 个均匀随机值，假设这些值的取值范围处于 0 和 1 之间，然后计算并打印它们的平均值（见练习 1.2.30）。
- 1.3.11 当你尝试打印标尺函数（见“使用 `while` 和 `for` 循环的典型示例”表格）时，如果使用的 n 值过大（如 100）会发生什么？
- 1.3.12 编写一个函数 `FunctionGrowth` 打印 $\log n$ 、 n 、 $n \log n$ 、 n^2 、 n^3 、 2^n 的取值表格，其中 n 的取值是 16、32、64、 \dots 、2048。使用制表符（`\t` 字符）来实现对齐。
- 1.3.13 执行以下代码后， m 和 n 的值是多少？

1.3.14 运行以下代码，将会打印出什么样的信息？

```
int f = 0, g = 1;
for (int i = 0; i <= 15; i++)
{
    System.out.println(f);
    f = f + g;
    g = f - g;
}
```

答案：即使是编程专家也会告诉你理解这样一个程序的唯一方法是对程序进行跟踪。当你这样做时，你会发现它的打印值为 0、1、1、2、3、5、8、13、21、34、55、89、144、233、377 和 610。这是著名的斐波那契数列的前 16 个，它的公式定义如下， $F_0=0$, $F_1=1$ ，当 $n>1$ 时 $F_n=F_{n-1}+F_{n-2}$ 。

82

1.3.15 以下代码段将会产生多少行输出？

```
for (int i = 0; i < 999; i++);
{ System.out.println("Hello"); }
```

答案：只打印一行。注意第一行末尾的分号。

1.3.16 编写一个程序，采用 int 型命令行参数 n 并打印所有小于或等于 n 并且是 2 的幂的正整数。确保你的程序对所有 n 的取值都是正确的。

1.3.17 拓展你的练习 1.2.24 程序，让它打印一张表格，列出最后支付的总金额和每个月还款后剩余的本金。

1.3.18 与谐波数不同，对于 $1/1^2+1/2^2+\dots+1/n^2$ 之和，当 n 增长到无穷大时，它会收敛到一个常数（这个常数是 $\pi^2/6$ ，所以这个等式可以用来估算 π 的值）。以下哪个 for 循环可以用于计算这个总和？假设 n 是初始值为 1000000 的 int 型变量，sum 是初始化为 0.0 的 double 型变量。

- a. for (int i=1; i<=n; i++) sum+=1/(i*i); b. for (int i=1; i<=n; i++) sum+=1.0/i*i;
c. for (int i=1; i<=n; i++) sum+=1.0/(i*i); d. for (int i=1; i<=n; i++) sum+=1/(1.0*i*i);

1.3.19 程序 1.3.6 实现了基于牛顿法找到 c 的平方根，解释它的工作原理。提示：牛顿法可以用来解决任何函数的求根问题，对于可导函数 $f(x)$ ，在 $x=t$ 处的切线斜率为 $f'(t)$ ，利用这个结果就可以求出切线方程，然后使用该方程找到切线与 x 轴相交的地方，然后，在每次迭代时，将估算值 t 替换为 $t-f(t)/f'(t)$ 。

1.3.20 开发一个程序，由命令行输入两个 int 型参数 n、k，用牛顿法求出 n 的 k 次方根（提示：请参考练习 1.3.19）。

1.3.21 以程序 Binary 为基础编写一个新程序 Kary，输入两个命令行参数 i 和 k，并将 i 转换为基数 k 的数制表示。假设 i 是 Java 中的 long 数据类型，k 是 2 到 16 之间的整数。对于大于 10 的基使用 A 到 F 来表示第 11 位到第 16 位数字。

83

1.3.22 编写一段代码，将整数 n 的二进制表示放入字符串变量 s 中。

答案：对于这一问题，Java 中有一个内置函数 Integer.toBinaryString(n)，但是练习的重点不是使用这个函数，而是如何实现这个函数。基于程序 1.3.7，我们可以得到解决方案：

```
String s = "";
int power = 1;
while (power <= n/2) power *= 2;
while (power > 0)
{
    if (n < power) { s += 0; }
    else { s += 1; n -= power; }
    power /= 2;
}
```

另一个简单的通过从右到左逐位输出的方案如下：

```
String s = "";
for (int i = n; i > 0; i /= 2)
    s = (i % 2) + s;
```

这两种方法都需要仔细研究才能明白它的原理。

- 1.3.23 为 Gambler 程序编写一个新的版本, 使用两个嵌套的 for 循环或者两个嵌套的 while 循环来实现, 以代替现在 for 循环中内嵌 while 循环的实现方式。
- 1.3.24 编写一个 GamblerPlot 程序, 通过每次下注后打印一行星号以表示赌徒手中的赌资, 一颗星代表赌徒手中有一美元, 跟踪并打印赌徒破产的模拟过程。
- 1.3.25 修改 Gambler 程序, 增加一个命令行参数 p 用于指定赌徒每次下注时获胜的概率 (假设整个赌博过程中概率固定)。使用你的程序尝试分析这种概率是如何影响获胜的机会和预期的投注次数的。实验时使用的 p 值应接近 0.5 (如 0.48)。
- 84 1.3.26 修改 Gambler 程序, 增加一个命令行参数用于指定每次愿意进行的投注次数, 在游戏结束时有三种可能性: 赌徒获胜、破产或者时间耗尽。在游戏结束时, 输出赌徒拥有的金额。可以试试使用你的程序计划你的下一次蒙特卡洛赌场之旅。
- 1.3.27 修改 Factors 程序, 对于相同的因子只打印一次。
- 1.3.28 对于 Factors 程序 (程序 1.3.9), 快速地运行几次实验以确定使用终止条件的影响。对于 ($\text{factor} \leq n / \text{factor}$) 或者是 ($\text{factor} < n$), 找到最大的 n , 使得在你输入一个 n 位数字时, 程序能够在 10s 内结束。
- 1.3.29 编写一个 Checkerboard 程序, 输入一个 int 型命令行参数 n , 使用一个嵌套的循环, 打印出一个 $n \times n$ 棋盘, 棋盘上的空格和星号交替出现。
- 1.3.30 编写一个 GreatestCommonDivisor 程序, 它使用 Euclid 算法找到两个整数的最大公约数 (gcd), 它是基于下面的规律进行迭代计算的: 假设 x 大于 y , 如果 y 可以整除 x , 那么 x 和 y 中的最大公约数是 y , 否则 x 和 y 的最大公约数与 $x \% y$ 和 y 的最大公约数相同。
- 1.3.31 编写 RelativelyPrime 程序, 输入一个 int 型命令行参数 n , 打印一个 $n \times n$ 的表格, 如果 i 和 j 的最大公约数是 1, 则在第 i 行和第 j 列上打印一个 “*” (我们称 i 和 j 是互素的), 否则在此位置上打印一个空格。
- 1.3.32 编写一个程序 PowersOfK, 输入 int 型命令行参数 k , 并打印 Java 的 long 数据类型中所有 k 的幂。注意: 常量 Long.MAX_VALUE 是长整型数据中最大的数值。
- 1.3.33 编写一个打印球体表面随机点 (a, b, c) 坐标的程序。为了产生这样的点, 请使用 Marsaglia 方法: 首先根据本节末尾的描述在单位圆上选择一个随机点 (x, y) 。然后, 设置 a 为 $2x\sqrt{1-x^2-y^2}$, b 为 $2y\sqrt{1-x^2-y^2}$, c 为 $1-2(x^2+y^2)$ 。

创新练习

- 1.3.34 拉马努金的士数。拉马努金 (Srinivasa Ramanujan) 是印度的数学家, 因为对数字的直觉而闻名。当英国数学家 G.H. Hardy 有一天来拜访他时, Hardy 表示出租车的号码是 1729, 是一个毫无意义的数字。Ramanujan 回答: “不, Hardy, 这是一个非常有意思的数字。它可以分解成两个数字的立方之和, 并且是有两种不同分解方法的最小自然数 ($1729=1^3+12^3=9^3+10^3$, 后来这类数称为的士数——译者注)。”通过编写一个程序, 输入一个 int 型的命令行参数 n , 并打印小于或等于 n 的的士数, 以证明拉马努金的结论是否正确。换句话说, 找到不同整数 a 、 b 、 c 和 d , 使得 $a^3+b^3=c^3+d^3$ 。你可能需要使用四个嵌套的循环。
- 1.3.35 校验和。国际标准书号 (International Standard Book Number, ISBN) 是一个 10 位数的编号, 是一本书的唯一编号。这个编号中, 最右侧的数字是校验位, 是可以通过其他 9 位数字唯一确定的。具体的计算方法是: 计算 $1d_1+2d_2+3d_3+\dots+10d_{10}$, 得到的结果必须是 11 的倍数 (这里的 d_i 表示从右侧开始的第 i 位数字)。 d_1 就是校验和数字, 它的取值范围为 0 到 10 之间的

任何值。ISBN 还规定使用字符“X”来表示 10。例如，对应于 020131452 的校验和数字是 5，因为 5 是能够让下式值为 11 的倍数，且处于 0 到 10 之间的 x 的唯一值。

$$10 \cdot 0 + 9 \cdot 2 + 7 \cdot 1 + 6 \cdot 3 + 5 \cdot 1 + 4 \cdot 4 + 3 \cdot 5 + 2 \cdot 2 + 1 \cdot x$$

编写一个程序，输入 9 位 int 型的命令行参数，计算校验和，并打印 ISBN 码。

- 1.3.36 计算素数。编写一个程序 PrimeCounter，输入一个 int 型命令行参数 n ，找到小于或等于 n 的所有素数。使用它打印出小于或等于 1000 万的素数。注意：如果不仔细设计求解的方法，你的程序可能无法在合理的时间得出结果！

- 1.3.37 二维无规则运动。二维无规则运动可以模拟粒子在网格中移动的行为，在每一个模拟的步骤中，粒子都会以 1/4 的概率向东、南、西、北四个方向移动，与之前的移动方向无关。编写一个 RandomWalker 程序，输入一个 int 型的命令行参数 n ，假设粒子起始时停在一个 $2n \times 2n$ 的正方形的中心，估算需要多长时间才能到达正方形的边界。

88

- 1.3.38 指数函数。假设 x 是一个 double 类型的正数变量。编写一段程序，根据泰勒级数展开式计算指数函数，计算式为 $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$ 。

答案：本练习的目的是让你考虑如何使用基本的运算类型实现类似 Math.exp() 的函数。尝试给出你自己的解决方案，然后与这里给出的方案进行比较。

我们首先考虑如何计算展开式中的一项。我们用变量 term 来表示一项，假设 x 和 term 都是 double 类型的变量， n 是一个 int 类型的变量，以下代码片段将用于计算 $\frac{x^n}{n!}$ ，并将结果保存在 term 中。计算方法非常直接，使用一个循环计算分子，另一个循环计算分母，然后相除得到结果：

```
double num = 1.0, den = 1.0;
for (int i = 1; i <= n; i++) num *= x;
for (int i = 1; i <= n; i++) den *= i;
double term = num/den;
```

一个更好的方法是只使用一个 for 循环：

```
double term = 1.0;
for (i = 1; i <= n; i++) term *= x/i;
```

除了更紧凑和简洁之外，后一种解决方案还有一些优点，因为它可以避免因大量计算造成的不准确，例如，像 $x=10$ 和 $n=100$ 这样的值，双循环的方法就会产生误差，因为 $100!$ 太大了，不能使用双精度浮点数来表示。

为了计算 e^x ，我们再使用一个 for 循环来嵌套之前的 for 循环：

```
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; n++)
{
    sum += term;
    term = 1.0;
    for (int i = 1; i <= n; i++) term *= x/i;
}
```

循环迭代的次数取决于每一项的值与累加和的相对大小。一旦 sum 的值不再变化，我们就退出循环（这个策略比使用循环、条件判断（term>0）更有效，因为它避免了大量的不会改变 sum 值的迭代）。这一部分代码是有效的，但是它的效率非常低，因为在外部的 for 循环的每一次迭代中，内部的 for 循环都会重新计算先前迭代中计算过的所有值。我们可以利用在前一循环迭代计算出的 term 值，使用一个 for 循环来解决问题：

87

```
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; n++)
{
    sum += term;
    term *= x/n;
}
```

- 1.3.39 三角函数。编写两个程序——Sin 和 Cos，它们使用泰勒展开式计算正弦和余弦函数。

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots, \cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

- 1.3.40 实验分析。通过实验来分析我们给出的 Math.exp() 的几种实现方法的相对性能。练习 1.3.38 中计算 e^x 的方法有几种：直接用嵌套 for 循环的方法、使用单个 for 循环的改进方法，以及在此基础上使用的改进的循环连续条件（term>0）的方法。通过改变命令行参数的方法来试错运行，对于这三种不同的实现方法，看看你的计算机在 10s 内能够完成计算的次数。

- 1.3.41 佩皮斯问题。在 1693 年，塞缪尔·佩皮斯 (Samuel Pepys) 问艾萨克·牛顿：掷 6 次骰子至少一次得到 1 和掷 12 次骰子至少两次获得 1，哪个可能性更大？编写一个程序，帮助牛顿快速找到答案。

- 1.3.42 游戏模拟。在游戏节目“Let's Make a Deal”中，一个参赛者会面对三扇门，只有一个门后面是一个有价值的奖品。在参赛者选择一扇门后，主持人打开其余两扇门中的一扇（当然打开的门中不会有奖品）。参赛者此时有机会转向另外一个未打开的门。参赛者应该这样做吗？按照直觉来看，参赛者初次选择的门和其他未打开的门都有可能包含奖品，两者的概率是相同的，因此没有动机去切换。编写 MonteHall 程序，模拟测试你的这个直觉。你的程序应该需要一个命令参数 n，然后使用两个策略（切换或者不切换）进行 n 次游戏，并打印每种策略成功的概率。

88

- 1.3.43 5 个数的中位数。编写一个程序，将 5 个不同的数作为命令行参数，并打印其中位数（有两个数字比它小，另外两个比它大）。附加要求：对于任何一组输入，程序中发生的比较都应小于 7 次。

- 1.3.44 排序三个数。a、b、c 和 t 都是 int 类型的变量，解释以下代码为什么能将 a、b 和 c 按照升序来排定。

```
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

- 1.3.45 混乱。用程序实现下面的人口增长简易模型，并将它应用于研究一个池塘里的鱼、测试试管中的菌类和其他类似情况。我们假设人口在 0（灭绝）到 1（可维持的最大人口）之间波动。如果在 t 时刻人口数量是 x ，我们假设在 $t+1$ 时刻人口数量是 $rx(1-x)$ ，参数 r 被称为繁殖率，用于控制人口增长的速率。从一个很小的人口数量开始，如 $x=0.01$ ，研究不同的 r 值在迭代的过程中对人口的影响。要想使人口稳定在 $x=1-1/r$ ， r 值应该是多少？当 r 为 3.5、3.8、5 时，人口数量会发生什么？说说你的见解。

- 1.3.46 欧拉幂方数和的猜想。在 1769 年，莱昂哈德·欧拉 (Leonhard Euler) 提出了广义的费马大定理，猜想认为至少需要 n 个 n 次幂数相加，它们的和才可能是一个 n 次幂数 ($n>2$)。编写一个程序来反驳欧拉的猜想（事实上这个猜想直到 1967 年才被推翻），使用五层嵌套循环找到 4 个整数，其 5 次方的和是另一个正整数的 5 次幂方。也就是说，找到 a 、 b 、 c 、 d 和 e 使得 $a^5+b^5+c^5+d^5=e^5$ 。使用 long 数据类型。

89

1.4 数组

在本节中，我们将介绍数据结构（data structure）的概念，并构造你的第一个数据结构——数组。数组的主要目的是便于存储和处理大量的数据。数组在许多数据处理任务中起着至关重要的作用，它们也可以与向量和矩阵相对应，向量和矩阵广泛用于科学和科学编程。我们将探讨 Java 数组中的基本属性，并使用大量例子来说明数组的使用方法。

数据结构是计算机中数据组织的一种方式（通常是为了节省时间或者空间）。数据结构在计算机编程中起着至关重要的作用，在本书的第 4 章中将会讨论各种经典的数据结构。

一维数组（也简称为数组）是一种数据结构，用于存储数值的一个序列，这个序列中值的类型都是相同的。我们将数组中的每一个值称为它的一个元素。我们使用索引来引用数组中的元素：如果一个数组中包含 n 个元素，那么我们将元素按从 0 到 $n-1$ 编号，以便我们可以用一个整数（取值范围在 0 到 $n-1$ 之间）作索引来明确指定其中的每一个元素。

二维数组是以一维数组为元素组成的数组。一维数组的元素是由一个整数索引的，因而二维数组的元素由一对整数索引：第一个索引指定行，第二个索引指定列。

通常，当我们有大量的数据需要处理时，我们首先把所有数据放到一个或者多个数组中。接下来，我们使用索引来引用每个独立的元素并处理数据。我们需要处理的数据可能有考试成绩、股票价格、DNA 链中的核苷酸或书中的字符等。在这些例子中都包含了大量相同类型的值，是数组的典型应用场景。我们在 1.5 节中讨论输入 / 输出，在 1.6 节中进行案例分析，在这些章节中我们都会对这些例子进行仔细分析。在本节中，我们通过探讨一些例子来展示数组的基本属性。我们的程序首先使用从实验研究中计算出的值来填充数组，然后再对其进行处理。

a	a[0]
	a[1]
	a[2]
	a[3]
	a[4]
	a[5]
	a[6]
	a[7]

一个数组

90

Java 中的数组 在 Java 程序中创建一个数组涉及三个不同的步骤：

- 声明一个数组。
- 创建一个数组。
- 初始化数组中的元素。

要声明一个数组，你需要为它指定一个名称并指明它所包含的数据类型。要创建一个数组，你需要指定它的长度（元素的数量）。要初始化一个数组，你需要给它的每个元素赋值。例如下面的代码创建了一个包含 n 个元素的数组，其类型是 `double` 型，其中的元素被初始化为 0.0：

```
double[] a;           // 声明数组
a = new double[n];    // 创建数组
for (int i = 0; i < n; i++) // 初始化数组
    a[i] = 0.0;
```

第一个语句是数组声明，它和基本类型变量的声明几乎一样，除了类型名称后面要有一对方括号，它用于表明正在声明的是一个数组。第二个语句是创建数组，它使用关键字 `new` 来分配内存，用于存储指定数量的元素。对于基本类型的变量不需要此操作，但是在 Java 中，除了基本类型外其余所有类型数据都需要这个操作（参见 3.1 节）。for 循环为数组中的每个元素都赋值为 0.0。我们通过将索引放入方括号中来引用数组中的每个元素：代码 `a[i]` 表示的是引用数组 `a[]` 中的第 i 个元素（在本书中，我们使用符号 `a[]` 来表示变量 `a` 是一个数组，但是在 Java 中我们不使用 `a[]`）。

使用数组的明显优点是一次可以定义许多变量，而不需要一个一个明确声明它们。例如，如果你想处理 8 个类型为 `double` 型的变量，你可以这样声明它们：

```
double a0, a1, a2, a3, a4, a5, a6, a7;
```

然后用 `a0`、`a1`、`a2` 等等来引用它们。以这种方式命名数十个单独变量已经非常麻烦，命名数百万个这样的变量就更无法应对了。使用数组可以解决这类问题，你可以通过 `double[] a=new double[n]` 这样的语句来声明 `n` 个变量，并且用诸如 `a[0]`、`a[1]`、`a[2]` 等等来引用它们。用这个方法可以很容易地定义几十个或几百万个变量。此外，因为你可以使用变量（或者其他表达式的运行时计算结果）作为数组的索引，所以就像上面那样的循环，可以用来处理任意多个元素。数组中的每一个元素都可以看作一个单独的变量，你可以在一个表达式中使用它们，也可以在赋值语句中的左侧使用它们。

[91]

下面是关于数组的第一个例子，我们使用数组来表示向量。我们会在 3.3 节详细介绍向量；就目前而言，你可以将向量看作一连串的数字。两个向量（长度相等）的点积是它们相应元素的乘积之和。向量可以用一维数组 `x[]`、`y[]` 来表示，如果每个向量的长度是 3，它们的点积可以表示为 `x[0]*y[0]+ x[1]*y[1]+ x[2]*y[2]`。如果每个数组的长度是 `n`，那么下面代码可以用来计算它们的点积：

```
double sum = 0.0;
for (int i = 0; i < n; i++)
    sum += x[i]*y[i];
```

可以看到，数组的使用非常方便，也正因为如此，数组被大量运用在各式各样的编程任务中。

i	x[i]	y[i]	x[i]*y[i]	sum
				0.00
0	0.30	0.50	0.15	0.15
1	0.60	0.10	0.06	0.21
2	0.10	0.40	0.04	0.25
				0.25

点积的计算过程

在随后的表格中有很多数组处理的代码示例，在本书的后面我们将探讨更多的例子，因为在很多应用程序中，数组在处理数据中都扮演了非常重要的角色。在讲解更复杂的例子前，我们首先介绍一些使用数组编程的重要特性。

索引由 0 开始。数组 `a[]` 中的第一个元素是 `a[0]`，第二个元素是 `a[1]`，以此类推。显然，如果第一个元素是 `a[1]`，第二个元素是 `a[2]`，这样看起来更自然，但索引以 0 开始具有一定的优点，并且已经成为现代编程语言中的使用惯例。如果忽略了这个惯例，通常会导致数据访问错位。这个错误很难避免，也很难调试，所以要小心！

数组长度。一旦你在 Java 中创建了一个数组，它的长度就是固定的。需要在运行时显式创建数组的原因是，Java 的编译器在编译时还不清楚需要为数组预留多少空间，因为数组的长度有时候是在运行时才能确定的。你不需要为整型（`int`）或浮点型（`double`）变量显式分配内存，因为它们的大小是固定的，并且在编译时是已知的。你可以使用代码 `a.length` 来获得数组 `a` 的长度。注意数组 `a[]` 中最后一个元素应该是 `a[a.length-1]`。为了方便，我们通常用一个整型变量 `n` 来保存数组的长度。

[92]

创建一个填充了随机数的数组	<pre>double[] a = new double[n]; for (int i = 0; i < n; i++) a[i] = Math.random();</pre>
打印数组值，每行一个元素	<pre>for (int i = 0; i < n; i++) System.out.println(a[i]);</pre>
找到数组中的最大值	<pre>double max = Double.NEGATIVE_INFINITY; for (int i = 0; i < n; i++) if (a[i] > max) max = a[i];</pre>
计算数组元素的平均值	<pre>double sum = 0.0; for (int i = 0; i < n; i++) sum += a[i]; double average = sum / n;</pre>
颠倒数组中元素的顺序	<pre>for (int i = 0; i < n/2; i++) { double temp = a[i]; a[i] = a[n-1-i]; a[n-1-i] = temp; }</pre>
将一个序列拷贝到另一个数组中	<pre>double[] b = new double[n]; for (int i = 0; i < n; i++) b[i] = a[i];</pre>

典型的数组处理代码（假设 a[] 是 n 个 double 类型值的数组）

数组的默认初始化。为了减少代码，我们经常利用 Java 的默认数组初始化语句来一次性完成数组的声明、创建和初始化操作。例如，以下语句等同于本节“Java 中的数组”下的代码：

```
double[] a = new double[n];
```

等号左边的代码是数组声明；右边的代码是数组创建。在这种情况下不使用 for 循环赋初始值也是可以的，因为 Java 会自动将任何基本类型的数组元素初始化为 0（对于数字类型）或 false（对于布尔类型）。Java 也会自动将 String 类型的数组元素（以及其他非基本类型）初始化为 null（null 是一个特殊的值，你将在第 3 章中学到它）。

93

内存表示。数组是一种基础数据结构，可以在计算机内存中找到直接的对应形式。一个数组的所有元素在内存中的存储是连续的，因此很容易快速访问数组中的值。事实上，我们可以将内存本身看作一个巨大的数组。在现代计算机中，内存硬件上就是一连串存储单元，每个单元都有一个对应的位置索引，可以通过这个位置索引作为索引来访问这个单元。当讨论计算机内存相关问题时，我们通常将这个位置索引称为存储单元的地址。对于一个数组 a，为了便于理解，我们可以认为 a 中存储的是数组第一个元素 a[0] 的内存地址。为了便于说明，假设计算机的内存是由 1000 个存储单元组成的，它们的地址分别是 000 到 999（这个简化的模型忽略了一个事实，即数组元素可以根据其类型占用不同大小的内存，但是现在可以忽略这样的细节）。现在假设一个由 8 个元素构成的数组被存储在内存地址 523 到 530 中。在这种情况下，Java 将会把第一个元素的内存地址（索引）与数组长度一起存储在内存的某处。我们将这个地址称为指针，并说它指向被引用元素的内存位置。当我们指定 a[i] 时，编译器会通过将索引 i 加上 a[] 的内存地址来实现对所需值的访问。例如，Java 代码 a[4] 将生成机器码，以查找内存位置 523+4=527 处的值。访问数组的第 i 个元素是一个高效的 操作，因为它只需要对两个整数值进行相加，然后访问对应的地址，仅仅是两个内存操作。

内存分配。当你使用关键字 `new` 来创建一个数组时，Java 会在内存中预留足够空间来存储指定数量的元素。这个过程称为内存分配。在程序中使用的任何变量都需要一个这样的过程（但是由于 Java 知道需要分配多少内存给基本类型变量，因此不需要使用关键字 `new` 来创建基本类型）。现在请注意，在访问数组的任何元素之前，你必须先创建这一个数组。如果你不遵守这个规则，你将在编译时遇到一个变量未初始化（uninitialized variable）的错误。

边界检查。使用数组编程的时候一定要当心。引用数组元素时，你有责任保证使用有效的索引来引用数组元素。如果你创建了一个长度为 n 的数组并使用值小于 0 或大于 $n-1$ 的索引，你的程序在运行时将会自动中止，并显示一个 `ArrayIndexOutOfBoundsException` 异常（这种错误叫作缓冲区溢出，在很多编程语言中，系统是不会检测这种错误的。这种未检测的错误可能是你调试过程的噩梦，但开发人员常常忽视这种错误并在已完成的程序中留下潜在的危险。你可能会惊讶地发现这样的错误可以被黑客利用来控制一个系统甚至个人计算机以传播病毒、窃取个人信息或者制造其他恶意破坏）。Java 提供的这种错误消息一开始可能会让你感到困扰，但是为了获取更安全的程序而付出这样的代价是值得的。

在编译时设置数组的值。若我们有少量的值需要在初始化时保存在数组中，那么可以在声明、创建和初始化的过程中在一对大括号之间列出这些值，并用逗号分开。我们可以在处理扑克牌的程序中使用以下代码：

```
String[] SUITS = { "Clubs", "Diamonds", "Hearts", "Spades" };

String[] RANKS =
{
    "2", "3", "4", "5", "6", "7", "8", "9", "10",
    "Jack", "Queen", "King", "Ace"
};
```

现在，我们就可以使用两个数组的组合来随机打印一张扑克名称了，代码如下：

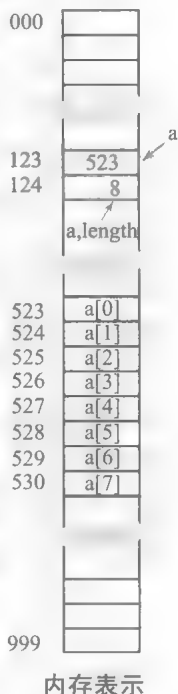
```
int i = (int) (Math.random() * RANKS.length);
int j = (int) (Math.random() * SUITS.length);
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

这段代码就可能显示出“Queen of Clubs”等。

这一段代码使用 1.2 节中介绍的方法来生成随机索引，然后使用索引从两个数组中挑选字符串。只要在编程时已经知道了所有数组元素的值（并且数组的长度不是太大），使用这种方式初始化数组是非常有效的，只需要在数组声明的过程中将所有值放在等号右边的大括号中间即可。这样做的过程同时也在暗示创建数组，因此也不需要关键字 `new`。

在运行时设置数组的值。更常见的情形是我们希望计算出数组中存储的值，而不是在编译前就确定下来。在这种情况下，我们可以使用赋值语句对数组中的任何一个元素赋值，赋值的过程就像其他赋值语句一样，只是语句的左侧是由数组名和索引组成的。例如，我们可以使用下面的代码来初始化一个长度为 52 的数组来代表一副扑克牌（使用刚刚定义的两个数组）：

```
String[] deck = new String[RANKS.length * SUITS.length];
for (int i = 0; i < RANKS.length; i++)
    for (int j = 0; j < SUITS.length; j++)
        deck[SUITS.length*i + j] = RANKS[i] + " of " + SUITS[j];
```



94

95

在这段代码被执行完毕以后，如果你按照 `deck[0]` 到 `deck[51]` 的顺序打印 `deck[]` 的内容，你会得到以下形式的输出结果：

```
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
...
Ace of Hearts
Ace of Spades
```

交换数组中的两个值。在编程时，有时我们会希望交换数组中的两个元素的值。继续使用扑克牌的例子，以下代码使用了我们在 1.2 节中使用过的方法来交换第 *i* 和第 *j* 处的扑克牌。

```
String temp = deck[i];
deck[i] = deck[j];
deck[j] = temp;
```

例如，如果我们对前面例子中的 `deck []` 数组运行这段代码，假设 *i* 等于 1、*j* 等于 4，那么运行之后它将使得 `deck[1]` 中存储着 “3 of Clubs”，在 `deck[4]` 中存储着 “2 of Diamonds”。你也可以试一下，当 *i* 和 *j* 相等时，这段代码运行后不会对数组的值产生任何改变。所以当我们使用这一部分代码时，我们只是在改变数组中值的顺序，而将数组作为一个集合来看时，它并没有发生变化。

96

数组元素混洗。以下代码可以实现对我们扑克牌数组中值的混洗：

```
int n = deck.length;
for (int i = 0; i < n; i++)
{
    int r = i + (int) (Math.random() * (n-i));
    String temp = deck[i];
    deck[i] = deck[r];
    deck[r] = temp;
}
```

我们每次从 `deck[i]` 到 `deck[n-1]` 中随机抽取一张扑克牌（每张扑克牌的可能性相同），将它与 `deck[i]` 中的值进行交换，这一操作对所有扑克牌从左到右依次执行。这一部分代码比所看起来的要复杂：首先，我们要保证混洗后和洗牌前的扑克牌是相同的；其次，我们每次都是从尚未选择的牌中来选择一张牌进行交换，并且选择的概率是均等的，以此来确保洗牌是随机的。

无须交换的取样。在许多情况下，我们想要从一个集合中随机抽取一个样本，且原来集合中的成员在样本中最多出现一次。很多游戏都是这样的采样过程，如从篮子里面抽取一个带数字的乒乓球进行抽奖，或者从一副扑克牌中抽取一手牌。程序 `Sample`（程序 1.4.1）展示了如何进行抽样，这里使用的核心方法和我们在实现混洗时使用的底层基本操作是一样的。它需要两个命令行参数 *m* 和 *n*，并创建一个长度为 *n* 的数列，数列中随机排列着从 0 到 *n-1* 的整数，每个数字只出现一次，将数列中的前 *m* 个元素作为一个随机采样的结果进行输出。为了便于理解这个处理过程，下面展示的是每个主循环的迭代结束时 `perm[]` 数组的内容（假设 *m* 和 *n* 的值分别是 6 和 16）。

i	r	perm[]															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	9	9	1	2	3	4	5	6	7	8	0	10	11	12	13	14	15
1	5	9	5	2	3	4	1	6	7	8	0	10	11	12	13	14	15
2	13	9	5	13	3	4	1	6	7	8	0	10	11	12	2	14	15
3	5	9	5	13	1	4	3	6	7	8	0	10	11	12	2	14	15
4	11	9	5	13	1	11	3	6	7	8	0	10	4	12	2	14	15
5	8	9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15
		9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15

运行命令“java Sample 6 16” 的追踪结果

97

程序1.4.1 无须交换的采样

```
public class Sample
{
    public static void main(String[] args)
    {
        // 打印m个整数的随机采样
        // 从0...n-1 (不重复)
        int m = Integer.parseInt(args[0]);
        int n = Integer.parseInt(args[1]);
        int[] perm = new int[n];

        // 初始化perm[]
        for (int j = 0; j < n; j++)
            perm[j] = j;

        // 采样
        for (int i = 0; i < m; i++)
        {
            // 在perm[i]的右边随机选一个数与之交换
            int r = i + (int) (Math.random() * (n-i));
            int t = perm[r];
            perm[r] = perm[i];
            perm[i] = t;
        }

        // 打印采样结果
        for (int i = 0; i < m; i++)
            System.out.print(perm[i] + " ");
        System.out.println();
    }
}
```

m	采样大小
n	范围
perm[]	0到n-1的排列

本程序需输入两个命令行参数m和n，并产生一个m个整数的随机采样，分布在0到n-1之间。这个过程不仅适用于州和地方彩票，也能用于各种科学应用中。如果第一个参数等于第二个参数，则输出结果是整数从0到n-1的随机排列。如果第一个参数大于第二个参数，程序将以ArrayOutOfBoundsException异常结束。

```
% java Sample 6 16
9 5 13 1 11 8
% java Sample 10 1000
656 488 298 534 811 97 813 156 424 109
% java Sample 20 20
6 12 9 8 13 19 0 2 4 5 18 1 14 16 17 3 7 11 10 15
```

98

在上面的程序中，假定变量 r 在给定范围内使取值的概率相等，那么在处理结束后 perm[0]

到 `perm[m-1]` 的值是一个均匀的随机采样（即使其中的一些元素可能被移动了多次），因为样本中的每个元素都是从那些尚未采样的值中随机选取一个值。计算这种随机采样数列的另一个重要用途是，可以把这个随机采样数列的值作为索引，用来实现对任意一个数组随机的采样。这么做的原因可能是，那个待采样的数组出于某种原因无法进行重新排列，在这种情况下，间接的进行采样就是一个不错的方案。（例如，某公司可能希望对客户列表抽取随机样本，而客户列表是按字母顺序保存的，不能被随意打乱。）

为了了解这个技巧是如何工作的，假设我们希望从我们的 `deck[]` 数组中随机抽取一张扑克牌，如上所述。我们使用采样代码并使 `n=52`、`m=5`，并在 `System.out.print()` 中将 `perm[i]` 替换为 `deck[perm[i]]`（将替换的结果打印出来），输出如下：

```
3 of Clubs
Jack of Hearts
6 of Spades
Ace of Clubs
10 of Diamonds
```

无论什么时候，只要我们想通过分析一个小的随机样本得出关于大群体的结论，都需要这样的随机抽样过程。随机采样已经广泛用作投票、科学研究和许多其他应用统计研究的基础。

预先计算的值。数组的一个简单应用是保存你已经计算出来的值，方便以后使用。例如，假设你正在编写一个使用一些谐波数执行计算的程序（见程序 1.3.5）。将这些值保存在数组中是一个有效的方案，如下所示：

```
double[] harmonic = new double[n];
for (int i = 1; i < n; i++)
    harmonic[i] = harmonic[i-1] + 1.0/i;
```

99

接下来你可以使用 `harmonic[i]` 这样的代码来引用第 `i` 个谐波数。以这种方式预先计算值是一个时间复杂度和空间复杂度的权衡：通过增加空间（保存值），我们节省了时间的开销（因为我们不需要重新计算值）。当 `n` 的值非常大时，这种方案并不是一个很有效的方案，但是，如果我们需要存储的值数量比较小，或者使用的次数很多时，则这是一个很有效的方案。

简化重复的代码。考虑下面的代码段，根据编号打印月份的名称（1 为 1 月份，2 为 2 月份，等等）：

```
if (m == 1) System.out.println("Jan");
else if (m == 2) System.out.println("Feb");
else if (m == 3) System.out.println("Mar");
else if (m == 4) System.out.println("Apr");
else if (m == 5) System.out.println("May");
else if (m == 6) System.out.println("Jun");
else if (m == 7) System.out.println("Jul");
else if (m == 8) System.out.println("Aug");
else if (m == 9) System.out.println("Sep");
else if (m == 10) System.out.println("Oct");
else if (m == 11) System.out.println("Nov");
else if (m == 12) System.out.println("Dec");
```

我们也可以使用 `switch` 语句，但是更简洁的选择是使用由每个月份的名称组成的字符数组：

```
String[] MONTHS =
{
    "", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
System.out.println(MONTHS[m]);
```

如果你在程序中的多个地方都需要通过数字来访问月份的名称，这种技术非常有用。注意我们必须在数组中浪费一个位置（元素 0），使得 MONTHS[1] 对应的是 1 月份。

有了这些基本的定义和例子，现在我们可以考虑两个应用程序，这两个程序都涉及有趣的经典问题，并且能够展示数组在高效计算中基本的重要作用。在这两个例子中，使用表达式的计算结果来索引数组中的元素起到了核心作用，而且如果不使用这个技术，这些计算会变得很困难，甚至是无法完成的。

100

卡券收集 假设你有一副扑克牌，并且你每次都均匀随机地从中抽出一张扑克牌（抽出的不再放回），你需要抽出多少张牌才能保证每种花色都被抽到过？你需要抽出多少张牌才能保证每种牌面的数字都被抽到过？这就是著名的卡券收集问题的例子。一般来说，假设卡券交易公司发行具有 n 种不同的卡券：你需要收集多少张卡才有可能收集到所有 n 种不同的卡券？假设你收集的每张卡的可能性相同。



卡券收集

卡券收集不是一个幼稚的问题。例如，科学家通常想知道某个自然界中出现的序列是否与某个随机序列具有相同的特征。如果是这样的话，这将会是一个非常有趣的事情（那就意味着在一些计算中自然界的序列和随机数序列可以相互替代——译者注）；如果不是，则需要进一步调查，以寻找数列的重要模式。例如，科学家使用这种方法来确定哪些部分的基因组序列值得研究。检测一个序列中的数字是否真的是随机的，一个有效的测试手段就是运行卡券收集测试，即比较找到所有可能的值所需要访问的元素数量是否与均匀的随机分布序列的数值相同。CouponCollector（程序 1.4.2）是一个模拟此过程的程序，同时我们用它来阐述数组的使用。程序需要输入一个命令行参数 n ，并使用代码 `(int)(Math.random() * n)`（见程序 1.2.5）产生一组 0 到 $n-1$ 的随机整数序列。假设每一个整数代表了一张牌，我们想知道我们是否已经见过此牌。为了方便，我们使用一个名为 `isCollected[]` 的数组，我们使用牌的值作为索引，如果第 i 张牌看过了，`isCollected[i]` 就是 `true`，否则就是 `false`。我们使用整数 r 来代表我们得到的一张新牌，我们通过访问 `isCollected[r]` 来检查我们是否之前见过这张牌。计算过程中需要记录看到的卡券的种类和生成的卡券的数量，并当前者数量到达 n 时打印出后者。

r	isCollected[]	distinct	count
	0 1 2 3 4 5		
	F F F F F F	0	0
2	F F T F F F	1	1
0	T F T F F F	2	2
4	T F T F T F	3	3
0	T F T F T F	3	4
1	T T T F T F	4	5
2	T T T F T F	4	6
5	T T T F T T	5	7
0	T T T F T T	5	8
1	T T T F T T	5	9
3	T T T T T T	6	10

101

程序1.4.2 模拟卡券收集（一）

```
public class CouponCollector
{
    public static void main(String[] args)
    {
        // 在[0..n)中生成随机数，直到找到每个种类
        int n = Integer.parseInt(args[0]);
        boolean[] isCollected = new boolean[n];
        int count = 0;
        int distinct = 0;
        while (distinct < n)
        {
            // 生成另一张卡券
            int r = (int) (Math.random() * n);
            count++;
            if (!isCollected[r])
            {
                distinct++;
                isCollected[r] = true;
            }
        } // 找到n个不同的卡券
        System.out.println(count);
    }
}
```

n	#卡券的值 (0~n-1)
isCollected[i]	卡券i是否被收藏过?
count	卡券的数量
distinct	不同卡券的数量
r	一个随机的卡券

这个程序需要在命令行中输入一个整数n作为参数，然后程序会生成0到n-1之间的随机数以进行卡券收集过程的模拟，直到收集所有可能的卡券类型时程序结束。

```
% java CouponCollector 1000
6583
% java CouponCollector 1000
6477
% java CouponCollector 1000000
12782673
```

与往常一样，理解程序最好的方式是跟踪一个典型的运行过程中变量值的变化。这个过程非常容易，向 CouponCollector 中添加一段代码，用来在每一个 while 循环结束时输出变量的值。在附图中为了便于显示，我们使用 F 代表 false，T 代表 true。跟踪使用庞大数组的程序时可能面临一个挑战：当程序中有一个长度为 n 的数组时，它代表有 n 个变量，你必须把它们全部列出来。跟踪使用 Math.random() 的程序也是一个挑战，因为每次运行程序时都会得到不同的跟踪结果。在这里，请注意 distinct 的值总是等于 isCollected[] 中 true 的数量。

如果没有数组，当 n 非常巨大时，我们无法想象如何完成模拟卡券收集程序的处理过程；而如果使用数组来实现就会变得非常方便。在本书中我们会看到很多这样使用数组的例子。

埃拉托斯特尼筛法 素数在数学、计算和密码学中扮演着重要的角色。一个素数是一个大于1的整数，其仅能被1和它本身整除。素数的计算函数 $\pi(n)$ 用于计算小于或等于 n 的素数个数。例如， $\pi(25)=9$ ，因为 25 之前包括 9 个素数，分别是 2、3、5、7、11、13、17、19 和 23。这个函数在数论中起到核心作用。

计算素数的一种方法是使用像 Factors（程序 1.3.9）这样的程序。具体来说，我们可以修改 Factors 中的代码，如果给定的数字是素数，则将一个布尔变量置为 true，否则设置为 false（而不再是把因子输出），然后将代码放到循环中，该循环每遇到一个素数计数器就递

增1, 这种方法对于 n 很小时非常有效, 但是对于 n 很大时运行会很缓慢。

PrimeSieve (程序 1.4.3) 需要输入一个命令行整数参数 n , 并使用埃拉托斯特尼筛法 (Sieve of Eratosthenes) 来找到素数的个数。该程序使用布尔数组 `isPrime[]` 来记录哪些整数是素数, 如果目标整数 i 是素数, 则将 `isPrime[i]` 设置为 `true`, 否则设置为 `false`。筛选的工作原理如下: 首先, 将所有数组设置为 `true`, 表示对于每一个整数都没有找到它的因子。然后, 从 $i=2$ 开始, 直到 $i \leq n/i$, 重复以下步骤:

- 找到下一个最小的素数 i (即没有找到其因子的整数)。
- 因为没有比 i 更小的因子, 设置 `isPrime[i]` 为 `true`。
- 将 i 的所有倍数对应的 `isPrime[]` 元素设置为 `false`。

当嵌套 `for` 循环结束时, 当且仅当整数 i 是素数时, `isPrime[i]` 为 `true`。然后我们遍历这个数组, 就可以计算小于或等于 n 的素数的个数。

程序1.4.3 埃拉托斯特尼筛法

```
public class PrimeSieve
{
    public static void main(String[] args)
    { // 打印小于或等于n的素数
        int n = Integer.parseInt(args[0]);
        boolean[] isPrime = new boolean[n+1];
        for (int i = 2; i <= n; i++)
            isPrime[i] = true;

        for (int i = 2; i <= n/i; i++)
        { if (isPrime[i])
            { // 将i的倍数标记为非素数
                for (int j = i; j <= n/i; j++)
                    isPrime[i * j] = false;
            }
        }

        // 统计素数个数
        int primes = 0;
        for (int i = 2; i <= n; i++)
            if (isPrime[i]) primes++;
        System.out.println(primes);
    }
}
```

n	参数
<code>isPrime[i]</code>	i 是否为素数?
<code>primes</code>	素数的个数

该程序需要输入一个整数型命令行参数 n 并计算小于或等于 n 的素数。为此, 使用布尔数组 `isPrime` 进行计算, 如果 i 是素数则设置 `isPrime[i]` 为 `true`, 否则设置为 `false`。首先, 它将数组的所有元素都设置为 `true` 来表示每个数字初始化时都被认为是素数。然后将非素数 (已知素数的倍数) 在对应数组中的元素设置为 `false`。如果 `a[i]` 在多轮将数组的元素设置为 `false` 的操作后还是 `true`, 那么我们可以知道 i 是素数。在第二个 `for` 循环中终止测试条件是 $i \leq n/i$ 而不是简单的 $i \leq n$, 因为如果一个整数没有小于 n/i 的因子, 那么它肯定不会有大于 n/i 的因子, 所以我们不必关注这样的因子。这种改进使得 n 较大时程序的运行时间大幅度缩短。

```
% java PrimeSieve 25
9
% java PrimeSieve 100
25
% java PrimeSieve 1000000000
50847534
```

像往常一样，可以很容易地通过添加代码来打印追踪。对于像 PrimeSieve 这样的程序，你必须要小心，因为它包含了一个嵌套 for-if-for，所以你必须注意花括号以把打印代码放在正确的位置。请注意，当 $i > n/i$ 时，我们就停止，就像我们在 Factors 中所做的那样。

		isPrime[]																								
1		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
		T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	
2		T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	
3		T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	
5		T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F	
		T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F	

运行命令 “java PrimeSieve 25” 的追踪信息

使用 PrimeSieve，我们可以计算大 n 的 $\pi(n)$ ，这时程序的限制变成了 Java 允许的最大数组长度。这是程序的时间和空间的典型例子。像 PrimeSieve 这样的程序在帮助数学家发展数字理论方面发挥了重要的作用，同时这个理论还有很多其他重要的应用。

105

二维数组 在很多应用程序中，一种方便的信息存储形式是将数字存储在规则的矩形表格中，并引用表格中的行和列来指定要访问的数据。例如，一个教师可能需要维护一个表格，行对应于学生、列对应于成绩；科学家可能需要维护实验数据表，其中行代表某一次实验、列代表实验结果数据；或者一个程序员可能需要一张表，这张表的每一个元素是一个像素对应的灰度值或颜色值，并把这张表用于显示某个图像。

与这些表格相对应的数学抽象是矩阵；对应的 Java 结构是一个二维数组。你可能已经遇到了许多矩阵和二维数组的应用，你也将将在科学、工程和计算应用中遇到许多其他应用程序，我们将在本书中以示例的方式进行阐述。与向量和一维数组一样，许多最重要的应用程序都涉及处理大量数据，等到我们在 1.5 节中介绍输入和输出以后，我们再考虑这些应用程序。

		a[1][2]		
第1行→	99	85	98	
	98	57	78	
	92	77	76	
	94	32	11	
	99	34	22	
	90	46	54	
	76	59	88	
	92	66	89	
	97	71	24	
89	29	38		
		↓ 第2列		

二维数组

二维数组的编程方法也非常简单，对原来的 Java 一维数组的代码进行扩展即可。在二维数组 `a[][]` 的使用过程中，通常使用 `i` 来代表行，用 `j` 来代表列，使用符号 `a[i][j]` 来表示一个元素；声明一个二维数组时，我们需要多加一对方括号；创建数组时，则在类型名后指定行的数量，紧接着行后指定列的数量：

```
double[][] a = new double[m][n];
```

我们把这样一个数组称为 $m \times n$ 的数组。按照惯例，第一维是行数，第二维是列数。与一维数组一样，Java 将数组中的所有数值初始为 0，将数组中的所有布尔值元素初始化为假。

默认初始化。二维数组的默认初始化非常有用，因为它比一维数组省去了更多的代码。以下代码等同于我们单行创建和初始化的习惯：

106

```
double[][] a;  
a = new double[m][n];  
for (int i = 0; i < m; i++)
```

```

{ // 初始化第1行
  for (int j = 0; j < n; j++)
    a[i][j] = 0.0;
}

```

这段将二维数组初始化为0的代码是多余的,但是嵌套 for 循环将元素初始化为其他值还是非常有用的。参考这段代码,可以构建出很多种访问或修改二维数组中每个元素的代码。

输出。对于很多二维的数组处理操作我们使用嵌套 for 循环。例如,以表格格式打印一个 $m \times n$ 的数组,我们可以使用以下代码:

```

for (int i = 0; i < m; i++)
{ // 打印第i行
  for (int j = 0; j < n; j++)
    System.out.print(a[i][j] + " ");
  System.out.println();
}

```

如果需要的话,我们可以通过添加代码,给输出的数据添加行号和列号(见练习 1.4.6)。Java 程序员在编程时,通常认为行是从0开始自上到下递增的,列号是从0开始自左到右递增的。

内存表示。Java 以数组的数组来表示二维数组,一个具有 m 行和 n 列的二维数组,实际上是一个长度为 m 的数组,其中每个元素又是一个长度为 n 的一维数组。在 Java 中的二维数组 $a[i][j]$ 可以使用代码 $a[i]$ 来引用第 i 行(这是一个一维数组),但是没有相对应的方法来引用第 j 列。

107

编译时设置数组元素的值。Java 可以在编译时初始化数组的值,初始化的方法就是紧跟在数组声明的后面列出元素的值。如果将一个二维数组看成普通的数组,每个元素占一行,那么每行元素都是一个一维数组。初始化一个二维数组,需要依次初始化这些一维数组,我们用大括号括起一系列数值来初始化每个一维数组,然后用逗号分隔每个一维数组,如右边代码所示。

电子表格。数组的一个常见用法是用电子表格程序(如 Excel 等——译者注)存储数字表格。例如,一个班有 m 个学生、每个学生有 n 项考试成绩,老师需要维护一个 $(m+1) \times (n+1)$ 的数组,其中最后一列为每个学生的平均考试成绩,最后一行为考试的平均成绩。尽管我们通常在专门的应用程序中进行这样的计算,但是还是值得研究数组处理的底层代码。为了计算每个学生成绩的平均值(每一行的平均值),需要将每一行元素的总和除以 n 。这样逐行的顺序处理矩阵元素的代码被称为行主序。相似的,计算每次考试的平均成绩(每一列的平均值),即每一列元素的和除以 m 。这样逐列的顺序处理矩阵元素的代码被称为列主序。

$a[i][j]$

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]
a[3][0]	a[3][1]	a[3][2]
a[4][0]	a[4][1]	a[4][2]
a[5][0]	a[5][1]	a[5][2]
a[6][0]	a[6][1]	a[6][2]
a[7][0]	a[7][1]	a[7][2]
a[8][0]	a[8][1]	a[8][2]
a[9][0]	a[9][1]	a[9][2]

$a[5] \rightarrow$

一个 10×3 的数组

```

double[][] a =
{
  { 99.0, 85.0, 98.0, 0.0 },
  { 98.0, 57.0, 79.0, 0.0 },
  { 92.0, 77.0, 74.0, 0.0 },
  { 94.0, 62.0, 81.0, 0.0 },
  { 99.0, 94.0, 92.0, 0.0 },
  { 80.0, 76.5, 67.0, 0.0 },
  { 76.0, 58.5, 90.5, 0.0 },
  { 92.0, 66.0, 91.0, 0.0 },
  { 97.0, 70.5, 66.5, 0.0 },
  { 89.0, 89.5, 81.0, 0.0 },
  { 0.0, 0.0, 0.0, 0.0 }
};

```

编译时初始化一个 10×3 的双精度数组

			第n列的 行平均数	
		n=3		
99.0	85.0	98.0	94.0	$\frac{92+77+74}{3}$
98.0	57.0	79.0	78.0	
92.0	77.0	74.0	81.0	
94.0	62.0	81.0	79.0	
99.0	94.0	92.0	95.0	
80.0	76.5	67.0	74.5	
76.0	58.5	90.5	75.0	
92.0	66.0	91.0	83.0	
97.0	70.5	66.5	78.0	
89.0	89.5	81.0	86.5	
91.6	73.6	82.0	第m行的 列平均数	
$\frac{85+57+\dots+89.5}{10}$				
m=10				

计算行的平均值

```
for (int i = 0; i < m; i++)
{ //计算第i行的平均值
  double sum = 0.0;
  for (int j = 0; j < n; j++)
    sum += a[i][j];
  a[i][n] = sum / n;
}
```

计算列的平均值

```
for (int j = 0; j < n; j++)
{ //计算第j列的平均值
  double sum = 0.0;
  for (int i = 0; i < m; i++)
    sum += a[i][j];
  a[m][j] = sum / m;
}
```

典型的电子表格的计算

108

矩阵操作。矩阵运算是科学和工程中的典型应用，二维数组可以用来表示矩阵，并用来实现各种数学运算。即使这样的处理通常是在专业的应用程序中完成的，但底层的计算也是值得理解的。例如，你可以按下面的方法进行 $n \times n$ 矩阵的加法：

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    c[i][j] = a[i][j] + b[i][j];
```

类似的，你也可以对两个矩阵做乘法，你可能已经学习过矩阵的乘法，但如果你不熟悉或者不记得矩阵的乘法，下面用于两个矩阵相乘的 Java 代码基本与数学定义是相同的。如果 $a[][]$ 和 $b[][]$ 的乘积记为 $c[][]$ ，那么它的每个元素 $c[i][j]$ 是矩阵 $a[][]$ 的第 i 行和矩阵 $b[][]$ 的第 j 列的点积。

$a[][]$

.70	.20	.10
.30	.60	.10
.50	.10	.40

$a[1][2]$

$b[][]$

.20	.30	.50
.10	.20	.10
.10	.30	.40

$b[1][2]$

$c[][]$

.90	.50	.60
.40	.80	.20
.60	.40	.80

$c[1][2]$

矩阵加法

```
double[][] c = new double[n][n];
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < n; j++)
  {
    // 第i行和第j列做点积
    for (int k = 0; k < n; k++)
      c[i][j] += a[i][k]*b[k][j];
  }
}
```

$a[][]$		$b[][]$	第2列 ↓		$c[1][2]=0.3*0.5$																		
<table><tr><td>.70</td><td>.20</td><td>.10</td></tr><tr><td>.30</td><td>.60</td><td>.10</td></tr><tr><td>.50</td><td>.10</td><td>.40</td></tr></table>	.70	.20	.10	.30	.60	.10	.50	.10	.40	← 第1行	<table><tr><td>20</td><td>.30</td><td>.50</td></tr><tr><td>10</td><td>.20</td><td>.10</td></tr><tr><td>10</td><td>.30</td><td>.40</td></tr></table>	20	.30	.50	10	.20	.10	10	.30	.40			$+0.6*0.1$
.70	.20	.10																					
.30	.60	.10																					
.50	.10	.40																					
20	.30	.50																					
10	.20	.10																					
10	.30	.40																					
					$+0.1*0.4$																		
					$=0.25$																		
				$c[][]$																			
				<table><tr><td>.17</td><td>.28</td><td>.41</td></tr><tr><td>.13</td><td>.24</td><td>.25</td></tr><tr><td>.15</td><td>.29</td><td>.42</td></tr></table>	.17	.28	.41	.13	.24	.25	.15	.29	.42										
.17	.28	.41																					
.13	.24	.25																					
.15	.29	.42																					

矩阵乘法

109

矩阵乘法的特例。矩阵乘法有两个非常重要的特例，这些特殊的情况发生在其中一个矩阵的维度是 1（这时也可以称它为一个向量）的时候。一个特例是矩阵 - 向量乘法，即一个 $m \times n$ 矩阵乘以一个 n 维列向量（一个 $n \times 1$ 的矩阵）将会得到一个 $m \times 1$ 的列向量（结果中的每一个元素是对应矩阵的行和向量的点积）。第二个特例是向量 - 矩阵乘法，我们将一个

行向量 ($1 \times m$ 的矩阵) 乘以一个 $m \times n$ 的矩阵得到一个 $1 \times n$ 的行向量 (结果中的每一个元素是操作数向量和一个对应矩阵的列的点积)。这些操作提供了一个简洁的方式来表示大量的矩阵运算。例如, 对于具有 m 行 n 列元素的电子表格中行的平均值计算相当于矩阵 - 向量乘法, 其中行向量有 n 个元素都是 $1/n$ 。同样的, 电子表格中列的平均值计算相当于向量 - 矩阵乘法, 其中列向量的每个元素的值都是 $1/m$ 。在本章的末尾的重要应用中, 我们还会讨论向量 - 矩阵乘法。

[110]

不规则数组。实际上, 并不要求二维数组中所有行都有相同的长度——具有不同长度的数组通常被称为不规则数组 (见练习 1.4.34 的应用程序示例)。针对不规则数组的处理代码需要格外注意。例如, 这段代码打印了不规则数组的内容:

```
for (int i = 0; i < a.length; i++)
{
    for (int j = 0; j < a[i].length; j++)
        System.out.print(a[i][j] + " ");
    System.out.println();
}
```

这部分代码能够测试你对 Java 数组的理解程度, 所以你应该花一定的时间来研究它。在本书中, 我们通常使用方阵或者矩阵, 其维数由变量 m 和 n 给定。而上面的代码中使用 $a[i].length$ 这种形式的代码来获取数组长度, 就说明被处理的数组一定是不规则数组。

多维数组。利用跟二维数组类似的拓展方法, 我们可以编写任意维度数组的代码。例如, 我们用以下代码声明和初始化了一个三维数组:

```
double[][][] a = new double[n][n][n];
```

对于每个元素, 我们可以用类似 $a[i][j][k]$ 进行索引。

二维数组为矩阵提供了一个自然的表示形式, 矩阵在科学、数学和工程学中无处不在。它们还提供了一个更自然的方式来组织大量的数据——电子表格和很多其他计算程序的关键组件。通过笛卡儿坐标、二维或三维数组也为物理世界的模型提供了基础。在本书中会有三个领域的示例程序涉及它们。

[111]

示例: 自回避的随机游走 (self-avoiding random walks) 假设你把你的狗放在了一个城市中心的大街, 所有的大街形成一个网格, 每条路看起来都差不多。我们假设有 n 个南北方向的街道, n 个东西方向的街道, 所有街道都是有规律的间隔, 并且完全相交形成一个格子模型。狗会尝试逃离城市, 在每个交叉路口随机选择方向前进, 但是它可以通过气味来避免走到已经走过的地方。但是, 这只狗可能会陷入死胡同, 这时除了重新回到某个路口别无选择。在这种情况下, 狗逃离城市的概率会有多大呢? 这个有趣问题是著名的自回避的随机游走模型的一个简单例子, 它在聚合物和统计力学的研究中有着重要的科学应用。例如, 你可以用这个模型模拟一个材料链的生成过程, 每次增长一点, 直到没有增长的可能。为了更好地理解这样的过程, 科学家们尝试理解自回避随机游走的特性。

矩阵 - 向量乘法 $a[i][j] * x[j]$ b[i]

```
for (int i = 0; i < m; i++)
{
    // 累加 x[] 到 b[i]
    for (int j = 0; j < n; j++)
        b[i] += a[i][j] * x[j];
}
```

a[i][j]		b[i]
99 85 98		94
98 57 78		77
92 77 76		81
94 32 11	x[]	45
99 34 22	[.33]	51
90 46 54	[.33]	63
76 59 88	[.33]	74
92 66 89		82
97 71 24		64
89 29 38		52

← 行平均值

向量 - 矩阵乘法 $y[i] * a[i][j]$ c[j]

```
for (int j = 0; j < n; j++)
{
    // y[] 和第 j 列的点积
    for (int i = 0; i < m; i++)
        c[j] += y[i] * a[i][j];
}
```

y[] [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1]

a[i][j]		c[j]
99 85 98		92
98 57 78		55
92 77 76		57
94 32 11		
99 34 22		
90 46 54		
76 59 88		
92 66 89		
97 71 24		
89 29 38		

← 列平均值

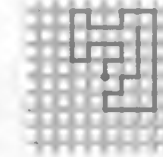
矩阵 - 向量和向量 - 矩阵乘法

狗逃跑的概率取决于城市的规模。在一个 5×5 的小城市里，很确定狗能够逃脱。但是当城市很大的时候能确保有机会逃脱吗？我们也对其他参数感兴趣。例如，狗为了逃走需要移动的平均距离是多少？这只狗在逃跑过程中走到某个特定区域的概率有多大？这些属性在刚刚提到的各种应用程序中很重要。

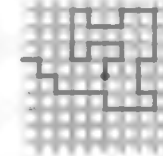
SelfAvoidingWalk (程序 1.4.4) 模拟了这种情况，它使用了一个二维布尔数组，其中每个元素表示一个交叉路口。值为 `true` 表示狗已经访问了该交叉路口；`false` 表示狗没有访问交叉路口。路径从中心开始，每次随机选择一个步骤，走到尚未访问的地方，直到在边界处成功逃跑或者被卡住。为了简单起见，代码的编写方式是，如果选择的随机步骤将会跳回到已经访问过的地点，则不会采取执行这个步骤的行动，而是重新再产生一个随机步骤以找到一个新的地点（在代码中，在这种情况下我们认为是走到了死胡同，并终止循环）。

请注意，该代码需要依赖于 Java 将每次实验的所有数组元素初始化为 `false`。它还展示了一个重要的编程技术，我们在 `while` 语句中编写循环退出测试，以防止循环体中出现非法语句。在这个例子中，`while` 循环连续条件可以用于防止循环内出界数组的访问。这同时也是在检查狗是否成功逃脱。在循环内部，如果检测发现是死胡同则会通过 `break` 跳出循环。

死胡同



逃脱



自回避游走

程序1.4.4 自回避的随机游走

<pre> public class SelfAvoidingWalk { public static void main(String[] args) { // 完成trials次随机自回避游走 // 在一个n×n的格子中走 int n = Integer.parseInt(args[0]); int trials = Integer.parseInt(args[1]); int deadEnds = 0; for (int t = 0; t < trials; t++) { boolean[][] a = new boolean[n][n]; int x = n/2, y = n/2; while (x > 0 && x < n-1 && y > 0 && y < n-1) { // 当前位置(x, y)是否已经访问过 a[x][y] = true; if (a[x-1][y] && a[x+1][y] && a[x][y-1] && a[x][y+1]) { deadEnds++; break; } double r = Math.random(); if (r < 0.25) { if (!a[x+1][y]) x++; } else if (r < 0.50) { if (!a[x-1][y]) x--; } else if (r < 0.75) { if (!a[x][y+1]) y++; } else if (r < 1.00) { if (!a[x][y-1]) y--; } } System.out.println(100*deadEnds/trials + "% dead ends"); } } } </pre>	<table border="1"> <tr><td>n</td><td>格子数目</td></tr> <tr><td>trials</td><td>实验次数</td></tr> <tr><td>deadEnds</td><td>实验陷入死胡同</td></tr> <tr><td>a[][]</td><td>访问过的交叉路口</td></tr> <tr><td>x, y</td><td>当前位置</td></tr> <tr><td>r</td><td>在(0, 1)中的随机数</td></tr> </table>	n	格子数目	trials	实验次数	deadEnds	实验陷入死胡同	a[][]	访问过的交叉路口	x, y	当前位置	r	在(0, 1)中的随机数
n	格子数目												
trials	实验次数												
deadEnds	实验陷入死胡同												
a[][]	访问过的交叉路口												
x, y	当前位置												
r	在(0, 1)中的随机数												

这个程序需要输入两个命令行参数 `n` 和 `trials`，然后在 $n \times n$ 的格子里面计算 `trials` 次自回避的随机游走。游走开始的位置在格子的中心，每走一步，创建一个布尔数组记录走过的位置，直到走进死胡同或者到达边界。最终计算的结果是死胡同情况占的百分比。增加实验数量可以提高精确度。

```

% java SelfAvoidingWalk 5 100
0% dead ends
% java SelfAvoidingWalk 20 100
36% dead ends
% java SelfAvoidingWalk 40 100
80% dead ends
% java SelfAvoidingWalk 80 100
98% dead ends

```

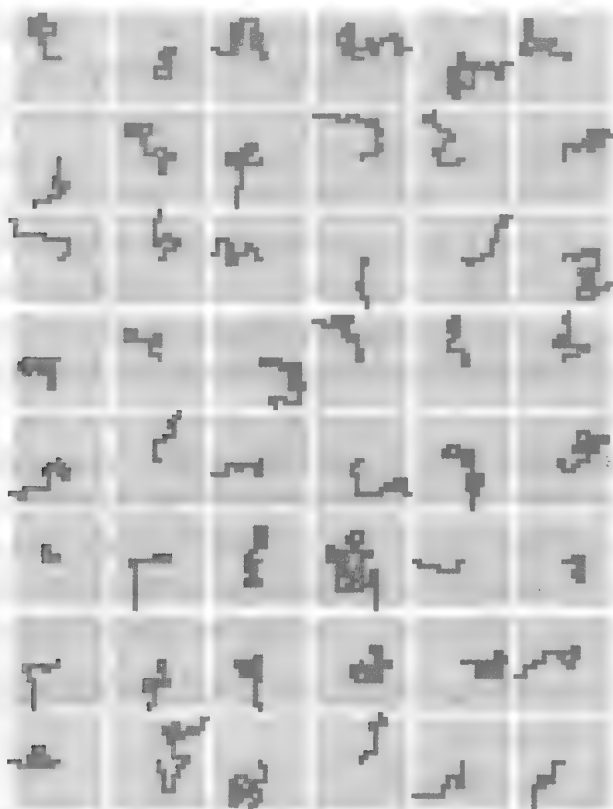
```

% java SelfAvoidingWalk 5 1000
0% dead ends
% java SelfAvoidingWalk 20 1000
32% dead ends
% java SelfAvoidingWalk 40 1000
70% dead ends
% java SelfAvoidingWalk 80 1000
95% dead ends

```

112

113

在 21×21 的格子里面的自回避随机游走

正如你在上文的例子中看到的那样，不幸的是，在一个足够大的城市网络中，你的狗会存在很大概率陷入死胡同，如果你对自回避游走有兴趣，可以在练习中找到更多的建议。例如，在三维空间里，狗逃脱的概率会大得多。这是一个从直观上很容易想到的结论，我们的测试也可以证明这个结论的正确性，但是自回避游走行为的数学模型仍然是一个未得到解决的开放问题；尽管进行了广泛的研究，但是仍然没有一个数学表达式能够用来描述逃脱的概率、路径的平均长度或任何其他重要的参数。

总结 数组是几乎所有编程语言中第四重要的基本元素（排在赋值语句、条件语句、循环语句之后）。完成了数组的学习，我们已经掌握了 Java 语言的基本结构。正如你在我们提供的示例程序中看到的那样，你可以使用这些结构编写程序来解决几乎所有的问题。

在我们学习过的许多程序中，数组的作用是很突出的，通过使用数组能够很好地解决许多编程任务。可能有的时候你没有显式地使用数组（其实你经常这样做），而是在“不经意间”隐式地使用了数组，因为所有计算机都有一片连续的内存，其概念上等同于数组。

数组作为一个基本元素添加到程序中，给我们带来的最大变化是会增加程序的状态数量。程序的状态可以定义为帮助你理解程序正在做什么所需要的信息。在没有数组的程序中，如果你知道了变量的值以及下一条要被执行的语句，通常可以确定程序接下来做什么。当我们追踪一个程序的执行过程时，我们基本上就是在追踪它的状态。然而当一个程序使用数组时，可能有太多的值（每一个值都可能被每一个语句所修改），使得我们不可能逐个跟踪它们。这种差异使得用数组编写程序比不使用数组更具有挑战。

数组能直接代表向量和矩阵，因此它们能直接用于与科学和工程中许多基本问题相关的运算。数组也提供了一种存储和处理大规模数据的标准方式，所以它们在任何涉及处理大数

据的应用程序中扮演着重要的角色，你在本书中会看到很多这样的例子。

115

问答环节

问：一些 Java 程序员使用 `int a[]` 而不是 `int[] a` 来声明一个数组。两者有何区别？

答：在 Java 中，两者都是合法的并且本质相同。前者是 C 中数组的声明方式，后者是 Java 中首选的方式，因为变量类型 `int[]` 更能清楚地说明是一个 `int` 数组。

问：为什么数组的索引是从 0 开始而不是从 1？

答：这个约定起源于计算机编程语言，其中数组元素的索引是通过数组的起始地址加上索引来计算的。从 1 开始索引将导致数组起始地址的浪费或者需要额外减 1 的时间。

问：如果我用一个负数来索引数组会发生什么？

答：结果与使用一个很大的数做索引一样。只要你尝试的索引值不在 0 和数组长度所规定的范围内，Java 就会抛出一个 `ArrayIndexOutOfBoundsException` 异常。

问：数组初始化时，为每一项设定的值是否必须是文字常量？

答：不必要。数组初始值中的条目可以是任意表达式（但必须与指定类型相匹配），即使它们的值在编译时还不能确定下来。例如下面的代码片段使用命令行参数 `theta` 初始化一个二维数组：

```
double theta = Double.parseDouble(args[0]);
double[][] rotation =
{
    { Math.cos(theta), -Math.sin(theta) },
    { Math.sin(theta),  Math.cos(theta) },
};
```

问：字符串 `String` 和字符数组之间有区别吗？

答：有。例如，你可以更改 `char[]` 数组中某一个特定的字符，但是不能更改 `String` 对象中的字符。我们将在 3.1 节中详细探讨字符串。

116

问：当使用 `(a==b)` 比较两个数组时会发生什么？

答：当且仅当 `a[]` 和 `b[]` 引用相同的数组（内存地址）时，表达式的计算结果为真值，这并不是用来判断它们存储的序列值是否相同。不幸的是，你通常并不是想比较它们引用的地址。因此，你需要用 `for` 循环来比较对应的元素是否相同来判断两个数组是否相同。

问：当为数组变量使用赋值语句时，如 `a=b`，会发生什么情况？

答：赋值语句能够使变量 `a` 和 `b` 引用相同的数组——但它不会按照你的预期把数组 `b` 中的值拷贝到数组 `a` 中。例如，考虑以下代码片段：

```
int[] a = { 1, 2, 3, 4 };
int[] b = { 5, 6, 7, 8 };
a = b;
a[0] = 9;
```

当使用了赋值语句 `a=b` 后，元素 `a[0]` 等于 5，元素 `a[1]` 等于 6，以此类推。也就是说两个数组对应相同的序列。然而它们并不是相互独立的数组。例如，在最后一个语句执行后，不仅仅是 `a[0]` 等于 9，`b[0]` 也等于 9。这是基本类型（如 `int` 和 `double`）和非基本类型（如数组）之间的重要区别，我们在学习 2.1 节中的将数组传递给函数和 3.1 中的引用类型时，将会更详细地回顾这个微妙的（但至关重要的）区别。

问：如果 `a[]` 是一个数组，为什么 `System.out.println(a)` 打印的是类似 `@f62373`，而不是数组中的值呢？

答：这是一个好问题，它打印的是数组的内存地址（以十六进制显示），不幸的是，这很少是你想要的。

问：使用数组时应该注意哪些其他陷阱？

答：记住创建数组时 Java 会自动初始化数组是非常重要的，因此创建数组花的时间与其长度成正比。

117

练习

1.4.1 编写一段代码，声明、创建和初始化一个长度为 1000 的数组 `a[]`，然后访问 `a[1000]`。你的程序能编译吗？当你运行时会发生什么？

1.4.2 描述并解释当你尝试使用下面语句编译程序时会发生什么情况：

```
int n = 1000;
int[] a = new int[n*n*n*n];
```

1.4.3 假设有两个长度为 n ，使用一维数组来表示的向量，编写一段代码，计算它们之间的欧氏距离（向量中对应元素的差的平方和，再求其平方根）。

1.4.4 编写一段代码，将一个一维字符数组中的值顺序颠倒过来，不要创建额外的数组来保存结果。提示：使用书中的代码来交换两个元素的值。

1.4.5 以下代码片段有什么错误？

```
int[] a;
for (int i = 0; i < 10; i++)
    a[i] = i * i;
```

1.4.6 编写一个代码段来打印二维布尔数组的内容，用 “*” 来表示真值，用空格来代表假值。需要打印出行号和列号。

1.4.7 以下代码段会打印什么？

```
int[] a = new int[10];
for (int i = 0; i < 10; i++)
    a[i] = 9 - i;
for (int i = 0; i < 10; i++)
    a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
    System.out.println(a[i]);
```

118

1.4.8 以下代码会在数组 `a[]` 中放入什么值？

```
int n = 10;
int[] a = new int[n];
a[0] = 1;
a[1] = 1;
for (int i = 2; i < n; i++)
    a[i] = a[i-1] + a[i-2];
```

1.4.9 以下代码段会打印什么？

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };
System.out.println(a == b);
```

- 1.4.10 编写一个程序 Deal，输入一个命令行参数 n，实现从一把洗好的扑克牌中抽出 n 手牌（每一手 5 张）并显示牌面的值，每手牌之间用空行分隔。
- 1.4.11 编写一个程序 HowMany，该程序可以接受任意多个命令行参数，并且打印出用户输入的命令行参数的个数。
- 1.4.12 编写一个程序 DiscreteDistribution，该程序可以接受任意多个整数型命令行参数，并且计算第 i 个命令行参数与 i 成比例的概率。
- 1.4.13 编写一段代码，创建一个二维数组 b[i][j]，并使它成为现存二维数组 a[i][j] 的副本，试针对下面每一种假设的情况进行处理：

a. [i][j] 是正方形 b. [i][j] 是矩形 c. [i][j] 可能是不规则的

b 的解决方案应该适用于 a，而 c 的解决方案应该适用于 b 和 a，并且你的代码应该逐渐变得更加复杂。

119

- 1.4.14 编写一个代码片段打印一个正方形二维数组的转置（行列交换）。例如，对于书中的电子表格数组，你的代码打印以下内容：

```
99 98 92 94 99 90 76 92 97 89
85 57 77 32 34 46 59 66 71 29
98 78 76 11 22 54 88 89 24 38
```

- 1.4.15 编写一个代码片段，在不需要创建第二个数组的情况下转置一个正方形的二维数组。
- 1.4.16 编写一个程序，需要输入整数型命令行参数 n，并创建一个 $n \times n$ 的布尔数组 a[i][j]，使得如果 i 和 j 互质（没有共同因子），a[i][j] 的值为真，否则为假。使用练习 1.4.6 的方案来打印数组。提示：使用素数筛法。
- 1.4.17 修改书中电子表格代码片段以计算行的加权平均值，其中每个考试成绩的权重在一维数组 weights[] 中。例如，要将这个例子中三次考试的最后一次的权重分配为前两次考试的两倍，你可以使用：

```
double[] weights = { 0.25, 0.25, 0.50 };
```

注意权重的和应该为 1。

- 1.4.18 写一段代码，实现两个不一定是正方形的矩形矩阵的乘法。注意：为了更好地定义点积，第一个矩阵中列的数量必须等于第二个矩阵中行的数量。如果不能满足此条件则打印一条错误消息。
- 1.4.19 编写一个程序，处理两个布尔方阵的乘法，使用 or 操作替代“+”，使用 and 操作替代“*”。
- 1.4.20 修改 SelfAvoidingWalk（程序 1.4.4），计算并打印路径的平均长度和走进死胡同的概率。分开计算逃跑路径的平均长度和死胡同路径的平均长度。
- 1.4.21 修改 SelfAvoidingWalk 以找到陷入死胡同的路径，并找出能够包含这一路径的最小轴对称矩形，计算这些矩形的平均面积。

120

创新练习

- 1.4.22 骰子模拟。下面的代码段计算两个骰子总和的确切概率分布：

```
int[] frequencies = new int[13];
for (int i = 1; i <= 6; i++)
    for (int j = 1; j <= 6; j++)
        frequencies[i+j]++;

double[] probabilities = new double[13];
for (int k = 1; k <= 12; k++)
    probabilities[k] = frequencies[k] / 36.0;
```

probabilities[k] 的值是骰子和为 k 的概率。通过运行试验模拟 n 次骰子投掷来验证此结论。假设每次投掷的结果均匀地分布在 1~6 之间的整数上, 计算这样的两个变量的和, 跟踪每个值的出现频率。n 需要多大, 才能使你的实验结果与上面给出的标准答案相匹配? (精度到小数点后三位时能够匹配即认为两个数字是匹配的。)

- 1.4.23 最长平台。给定一个整数数组, 找到最长平台的长度和位置。平台是指连续的等值序列, 并且紧接在该序列之前和之后的元素值较小。
- 1.4.24 经验混洗检验。运行计算实验来检查我们的混洗代码是否如我们预期的那样完成了它的工作。编写一个程序 ShuffleTest, 它需要输入两个整数命令行参数 m 和 n, 数组 a[] 大小为 m, 对数组 a[] 进行 n 次混洗, 每次混洗前数组 a[] 初始化为 a[i]=i。输出 m×m 的表格, 其中第 i 行存储的是 i 出现在位置 j 的次数。对于随机的数据, 数组中所有元素的结果都应趋近于 n/m。
- 1.4.25 不良混洗。假设我们在混洗代码中使用范围为 0 到 n-1 的随机数代替范围 i 到 n-1 之间的随机数。证明由此产生的序列在所有可能的 n! 种结果中出现的概率不是均等的。针对这一版本, 运行上一题的测试并观察结果。
- 1.4.26 音乐随机播放。将你的音乐播放器设置为随机播放模式。随机播放 n 首歌曲各一次, 然后重复。编写一个程序来估计你不会听到任何连续的两首歌曲的可能性 (也就是说, 歌曲 3 不跟随歌曲 2, 歌曲 10 不跟随歌曲 9, 等等)。
- 1.4.27 最小值的排列。编写一个采用整数命令行参数 n 的程序, 生成一个随机排列, 打印排列, 并打印排列中的从左到右的最小值的数目 (即到目前为止元素是最小的次数)。然后编写一个程序, 该程序需要两个整数命令行参数 m 和 n, 生成长度为 n 的 m 个随机排列, 并在生成的排列中打印从左到右的最小值的平均数。额外鼓励: 创建一个函数, 函数参数为 n, 返回数组大小为 n 的从左到右的最小值的数量。
- 1.4.28 倒序排列。编写一个程序, 从 n 个命令行参数读取从 0 到 n-1 的一个排列, 输出其倒序排列 (如果排列是一个数组 a[], 则其倒序排列 b[] 满足 a[b[i]]=b[a[i]]=i)。请确保输入参数为一个有效的排列。
- 1.4.29 哈达玛矩阵。 $n \times n$ 的哈达玛矩阵 $H(n)$ 是一个布尔矩阵, 具有显著的性质, 即任意两行在 $n/2$ 值中都不相同 (此性质可充分用于设计纠错码)。 $H(1)$ 是一个 1×1 的矩阵仅有一个值为真的元素。对于 $n > 1$ 的情况, $H(2n)$ 在一个大的正方形中包含了四个对齐的 $H(n)$ 的副本, 然后颠倒右下角的 $n \times n$ 的副本, 如以下例子所示 (其中, 通常情况下 T 代表真值, F 代表假值)。

$H(1)$	$H(2)$	$H(4)$
T	T T	T T T T
	T F	T F T F
		T T F F
		T F F T

122 编写一个采用整数命令行参数 n 的程序并打印 $H(n)$ 。假定 n 是 2 的幂。

- 1.4.30 谣言。Alice 正在与其他 n 个客人 (包括 Bob) 开一个派对。Bob 开始先制造了一个关于 Alice 的谣言, 并告诉其中一个客人。第一次听到这个谣言的客人会立即告诉下一个客人, 这个客人是从除了 Alice 和听到的人以外随机挑选出来的。如果一个人 (包括 Bob) 第二次听到了这个谣言, 则他 / 她不会进一步散播这个谣言。请编写程序以估计派对上的每个人 (除了 Alice) 都听到这个谣言的概率, 同时估计听到这个谣言的客人的数量。
- 1.4.31 素数的个数。比较程序 PrimeSieve 的方法和我们在 1.3 节最后阐述的 break 语句中使用的方法。这是一个典型的时空折中的例子: PrimeSieve 速度比较快, 但是要求使用一个长度

为 n 的布尔数组，另一种方法仅仅使用两个 `int` 变量，但实际上速度较慢。估计在与“java PrimeSeive 1000000”相同时间内，第二种方法可以找到的最大 n 值为多少？

- 1.4.32 扫雷游戏。编写一个程序，采用三个命令行参数 m 、 n 和 p ，并生成一个 $m \times n$ 的布尔数组，各元素的占用概率是 p 。在扫雷游戏中，占用状态单元格代表地雷，空单元格代表安全单元格，输出数组，使用星号表示地雷，使用英文句号为安全格。然后，用相邻炸弹的数量（上、下、左、右或对角线）创建一个整数二维数组。

```

* * . . .      * * 1 0 0
. . . . .      3 3 2 0 0
. * . . .      1 * 1 0 0

```

编写代码，以便通过使用 $(m+2) \times (n+2)$ 布尔数组来处理尽可能少的特例。

123

- 1.4.33 重复值查找。给定一个长度为 n 的整数数组，其中每个值都位于 1 到 n 之间，编写一个代码段来确定是否有重复的值。你可以不使用额外的数组（但你不必保留给定数组的内容）。
- 1.4.34 自回避行走的长度。假设网格的大小没有限制。运行实验估计平均路径长度。
- 1.4.35 三维自回避行走。运行实验以验证三维的自回避行走遇到死胡同的概率为 0，并计算 n 个不同值的平均路径长度。
- 1.4.36 随机步行者。假设 n 个随机步行者从 $n \times n$ 的网格中心开始，每次移动一步，每一步选择走上、下、左、右的概率是相同的。编写一个程序帮助制定和测试关于所有单元格被走过前走过的总步数的假设。
- 1.4.37 桥牌一手牌统计。在桥牌游戏中，四个选手各有一手 13 张扑克牌。其中一个重要的数据统计是每手牌的不同花色纸牌的数量。请问，5-3-3-2、4-4-3-2 或 4-3-3-3 哪个最有可能出现？
- 1.4.38 生日问题。假设人们进入一个空房间，直到有一对人的生日相同为止。平均进入多少人才能使得两个人生日相同？运行实验以估计此数量的值。假设生日在 0 到 364 之间均匀随机分布。
- 1.4.39 卡券收集问题。运行实验来验证经典的数学结果，即要收集 n 个不同类型的卡券需要购买卡券数量大约为 nH_n ，其中 H_n 是第 n 个谐波数。例如，如果仔细观察二十一点牌桌上的扑克（假设庄家洗好的牌数量足够多），平均意义上说，你需要翻开大约 235 张牌，才可以看见每张扑克牌。
- 1.4.40 鸽尾式洗牌。编写一个程序，使用鸽尾式洗牌法的 Gilbert-Shannon-Reeds 模型重新排列一副 n 张的扑克牌。首先，根据二项分布产生一个随机整数 r （假设把一枚完全对称的硬币随机抛起 n 次，其中正面朝上的次数为 r ）；其次，把一副牌一分为二，一半包括 r 张扑克牌，另一半包括 $n-r$ 张扑克牌。完成洗牌，重复下列步骤：从两堆牌中，把顶部的纸牌放在一堆新纸牌的下面。如果第一堆牌剩下 n_1 张牌，第二堆剩下 n_2 张牌，则从第一堆牌选取一张牌的概率为 $n_1/(n_1+n_2)$ ，从第二堆牌选取一张牌的概率为 $n_2/(n_1+n_2)$ 。研究需要多少次鸽尾式洗牌，才能把 52 张牌洗成一副均匀混洗的扑克牌。
- 1.4.41 二项式系数。编写一个程序，它需要一个整数命令行参数 n 并创建一个二维不规则数组 `a[][]`，其中 `a[n][k]` 包含了当你抛出一个硬币 n 次时恰好是 k 面的概率。这些数据称为二项分布：如果将第 i 行中的各元素乘以 2^n ，得到 $(x+1)^n$ 中 x^k 的系数的结果为按帕斯卡三角形排列的二项式系数。计算方式如下：首先，从对于所有的 n 和 `a[n][0]=0.0`、`a[1][1]=1.0` 开始；然后按行从左到右依次计算所得值，计算公式为：`a[n][k]=(a[n-1][k]+a[n-1][k-1])/2.0`。结果示意如表所示。

124

帕斯卡三角形	二项分布
1	1
1 1	1/2 1/2
1 2 1	1/4 1/2 1/4
1 3 3 1	1/8 3/8 3/8 1/8
1 4 6 4 1	1/16 1/4 3/8 1/4 1/16

125

1.5 输入 / 输出

在本节之前，我们一直使用命令行参数和标准输出作为我们的 Java 程序与外部交互的接口。在本节中，我们将扩展它们，以使得 Java 程序与外部的交互更加方便。这些新的接口包括标准输入（standard input）、标准绘图（standard drawing）和标准音频（standard audio）。标准输入可以用于处理任意数量的输入数据并实现与程序的交互；标准绘图可以编码图像；标准音频可以编码声音，使输入和交互不再局限于文本信息。这些新功能其实非常易于使用，并将带你进入一个编程的新境界。

I/O 通常指输入 / 输出（Input/Output 的英文首字母），它表达的意思也是这两个术语的组合。I/O 是程序与外部世界交流的机制。计算机操作系统控制着与计算机连接的各种物理设备，从而实现与外界的交互。我们的程序会使用一些 I/O 相关的库函数方法，这些方法会调用操作系统提供的接口最终实现 I/O。为了提高编程的通用性和便捷性，这些库函数通常将 I/O 操作进行标准化抽象（无论操作系统的实现如何，这些接口库会根据这些差异给出不同的实现，并提供一致的库函数接口——译者注）。

你已经学会了如何从命令行接收参数，如何在终端窗口打印字符串；本节将带你学习更多的数据处理和数据呈现工具。我们将要学习的这些工具和函数与前面学过的 `System.out.print()` 和 `System.out.println()` 库方法类似，这些函数实现的不是纯数学功能，而是实现一些对输入设备或输出设备的控制功能。我们会通过控制这些设备实现程序的数据输入和输出。

从程序的角度来看，标准 I/O 机制的一个基本特征是输入或输出的数据量没有限制。你的程序可以无限地消耗输入数据或者产生输出数据。

标准 I/O 机制的另一个用处是将程序连接到计算机外部存储中的文件（file）上。标准输入、标准输出、标准绘图和标准音频都可以很容易地连接到文件上，这使得 Java 程序很容易从文件中加载数据，或者将处理结果保存到文件中，以便存档或者供其他程序使用。

[126]

总览 从 1.1 节开始，我们一直在使用 Java 编程的常规模型。为了照应上下文，我们首先简要回顾一下这种模型。

Java 程序总是从命令行接收输入字符串，然后打印字符串作为输出。一般情况下，使用命令来运行的应用程序（也就是你用 `javac` 和 `java` 命令编译和执行的那个程序）都会用到命令行参数（command-line argument）和标准输出（standard output）。我们使用终端窗口（terminal window）来调用此应用程序。这种模型已被证明是一种方便直接地与我们的程序和数据进行交互的方式。

命令行参数。命令行参数是我们在编程过程中获取输入信息的常用方式之一，是 Java 编程的标准组成部分。任何类的 `main()` 方法都会有一个 `String` 数组 `args[]` 作为参数，该数组是由操作系统提供给 Java 的命令行参数序列。按照惯例，Java 和操作系统都将参数作为字符串进行处理，所以如果我们打算将一个参数用作一个数字，我们使用 `Integer.parseInt()` 或 `Double.parseDouble()` 这样的函数将它从 `String` 类型转换为相应类型。

标准输出。对于打印输出值，我们一直使用的是系统方法 `System.out.println()` 和 `System.out.print()`。Java 程序对这些方法的一系列调用，最终会生成抽象的字符流，我们把这个字符流称为标准输出（standard output）。默认情况下，操作系统将标准输出连接到终端窗口。到目前为止，我们程序中的所有输出都出现在终端窗口中。

`RandomSeq` 程序（程序 1.5.1）就是一个使用此模型的程序。它需要输入一个命令行参数 `n`，并产生分布在 0 和 1 之间的 `n` 个随机数。

程序1.5.1 生成一个随机序列

```
public class RandomSeq
{
    public static void main(String[] args)
    { // 打印[0,1)之间的n个随机实数的序列
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i < n; i++)
            System.out.println(Math.random());
    }
}
```

该程序使用的还是到目前为止我们一直在使用的Java编程常规模型。它需要输入一个命令行参数n，并打印处于0.0和1.0之间的n个随机数。从程序的角度来看，输出序列的长度没有限制。

```
% java RandomSeq 1000000
0.2498362534343327
0.5578468691774513
0.5702167639727175
0.32191774192688727
0.6865902823177537
...
```

现在我们将在命令行参数和标准输出的基础上扩充三个新的机制来解决它们的局限性，并为我们提供一个更有用的编程模型。这些机制为我们提供了一个全新的视角来理解 Java 程序如何将标准输入流和一系列命令行参数作为信息来源，以及如何将信息转换为标准输出流、标准图形流和标准音频流。

标准输入。我们要学习的类 `StdIn` 是一个基于标准输入流抽象接口的扩展库。就像你可以在程序中随意打印一个值到标准输出一样，通过这个库，你可以在程序执行期间从标准输入流中随时读取一个值。

标准绘图。我们要学习的类 `StdDraw` 可以实现在程序中创建图形。例如，可以在计算机窗口创建包含点和线的简单图形，也可以创建包含文本、颜色和动画等的复杂图形。

标准音频。我们学习的类 `StdAudio` 可以实现在程序中创建音频。它使用标准格式将数字序列转换为音频。

在使用命令行参数和标准输出的过程中，其实是我们一直在使用一些简单的 Java 内置功能，Java 还有一些更加抽象的内置工具，如标准输入、标准绘图和标准音频，但因为它们使用起来较为复杂，因此我们通过 `StdIn`、`StdDraw` 和 `StdAudio` 库提供了一些更简单的接口。同时，为了使我们的编程模型在逻辑上更加完整，我们还引入一个 `StdOut` 库。要使用这些库，你必须为 Java 提供 `StdIn.java`、`StdOut.java`、`StdDraw.java` 和 `StdAudio.java` 这几个源代码文件（有关详细信息，请参阅本节末尾的问答环节）。

标准输入和标准输出可追溯到 20 世纪 70 年代 UNIX 操作系统的开发，并且在所有现代系统中都以某种形式被实现。虽然它们比起那时开发的各种机制来说是基础的，但现代程序员仍然依赖它们作为将数据连接到程序的可靠方式。我们已经为本书开发了与早期抽象采用相同思想的标准绘图和标准音频，以提供一个简单的方法来产生视频和音频输出。

标准输出 Java 的 `System.out.print()` 和 `System.out.println()` 方法已经可以实现基本标准

输出，并且基本满足我们编程的各类需求。但是，为了使得输入和输出的编程模式统一，从本节开始及后面章节，我们将使用自定义的库 `StdOut`，其中也有与已使用的 Java 方法类似的函数，如 `StdOut.print()` 和 `StdOut.println()`（有关不同之处的讨论请参阅本书官网）。本节主要介绍 `StdOut.printf()` 方法，这个函数值得注意的一点是，它可以更好地控制输出的形式。这个功能最早在 20 世纪 70 年代初的 C 语言中引入，到目前这个功能仍然出现在很多现代语言中，足见其非常有用。

自从我们第一次输出 `double` 类型的值时，输出的精度问题一直困扰着我们。例如，当我们使用 `System.out.print(Math.PI)` 时我们得到的输出是 3.141592653589793，其实我们更希望得到的是 3.14 或者 3.14159。`print()` 和 `println()` 方法能够打印出数字的前 15 个小数位，但是有时我们不需要那么多位数。`printf()` 方法的功能则更加灵活。例如，它允许我们指定将浮点数转换为字符串输出时的小数位数。我们可以通过 `StdOut.printf("%7.5f", Math.PI)` 得到 3.14159，在 Newton（程序 1.3.6）中我们可以用。

```
StdOut.printf("The square root of %.1f is %.6f", c, t);
```

替换 `System.out.print(t)`，得到如下输出：

```
The square root of 2.0 is 1.414214
```

接下来，我们将描述这些语句的含义和操作，以处理其他内置数据类型。

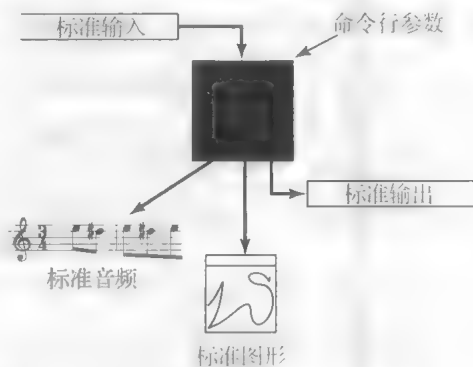
```
public class StdOut
```

<code>void print(String s)</code>	将s打印到标准输出
<code>void println(String s)</code>	打印s到标准输出并另起一行
<code>void println()</code>	在标准输出中另起一行
<code>void printf(String format, ...)</code>	根据格式规范将参数打印到标准输出

我们的库函数 API 中提供的用于标准输出的静态方法

格式化打印的基础知识。`printf()` 的最简单形式只有两个参数。第一个参数叫格式化字符串（format string）。它包含一个转换规范（conversion specification），描述如何将第二个参数转换为输出的字符串。转换规范的格式为 `%w.pc`，其中 `w` 和 `p` 是整数，`c` 是一个字符，这三个变量的具体解释如下：

- `w` 是字段宽度（field width），表示应写入的字符数。如果要写入的字符数超过（或等于）字段宽度，则忽略字段宽度；否则，输出时用空格在左边填充。负字段宽度表示输出应该用空格在右边填充。
- `.p` 是精度（precision）。对于浮点数，精度是小数点后的位数；对于字符串，它是应该打印的字符串的字符数。精度对整数没有意义。
- `c` 是转换码（conversion code）。我们最常使用的转换码是 `d`（用于 Java 整数类型的十进制值）、`f`（用于浮点数）、`e`（使用科学计算符号的浮点数）、`s`（用于字符串）和 `b`（用于布尔值）。



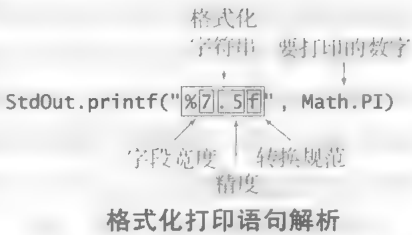
一个 Java 程序的总览图（回顾）

129

130

字段宽度和精度可以省略，但每个转换规范都必须具有转换码。

使用 printf() 最重要的一点是转换码和相应参数的类型必须匹配。也就是说，Java 必须能够从参数当前的类型转换为转换码所需的类型。每种类型的数据都可以转换为 String，但如果你运行 StdOut.printf("%12d", Math.PI) 或 StdOut.printf("%4.2f", 512)，则会得到一个运行时错误 `IllegalFormatConversionException`。



格式化字符串。除了转换规范之外，格式化字符串还可以包含字符。转换规范由参数值（转换为指定的字符串）替换，所有剩余字符均直接传递到输出。例如下面的代码。

```
StdOut.printf("PI is approximately %.2f.\n", Math.PI);
```

将会打印出一行

```
PI is approximately 3.14.
```

值得注意的是，我们需要在格式化字符串中明确包含换行符 `\n`，以实现通过 `print()` 打印新行。

多个参数。`printf()` 方法也可以使用两个以上的参数。在这种情况下，格式化字符串将为每个参数提供对应的转换规范，转换规范在格式化字符串中可能由其他字符分隔，以便在输出时能够有所区分。例如，计算贷款所支付的利息（见练习 1.5.13），则可以在内部循环中使用下列语句，用于打印表格中第二行和后续行：

```
String formats = "%3s  $%6.2f  $%7.2f  $%5.2f\n";
StdOut.printf(formats, month[i], pay, balance, interest);
```

131

就会得到下面这样的表格：

	payment	balance	interest
Jan	\$299.00	\$9742.67	\$41.67
Feb	\$299.00	\$9484.26	\$40.59
Mar	\$299.00	\$9224.78	\$39.52
...			

格式化打印很方便，因为这种代码比我们用来创建输出字符串的字符串连接代码要简洁。我们只描述了基本选项，更多信息可以参阅本书官网。

类型	代码	典型值	格式化字符串样例	输出的转换后字符串数值
int	d	512	"%14d"	" 512"
			"%-14d"	"512 "
double	f	1595.1680010754388	"%14.2f"	" 1595.17"
			"%.7f"	"1595.1680011"
			"%14.4e"	" 1.5952e+03"
String	s	"Hello, World"	"%14s"	" Hello, World"
			"%-14s"	"Hello, World "
			"%-14.5s"	"Hello "
boolean	b	true	"%b"	"true"

printf() 的格式约定（对于更多其他选项，请参见本书官网）

标准输入 我们的 StdIn 库从标准输入流中获取数据，该数据可能为空，也可能包含以空格（多空格、制表符、换行符等）分隔的值序列，但每个值都是一个字符串或者是一个来自 Java 某个基本类型的值。标准输入流的重要特点之一是：程序在读取数据时会消耗（consume）输入流中的值。一旦程序读取了一个值，它将无法备份并再次读取。这个假设是限制性的，但它反映了一些输入设备的物理特性。下文列出了 StdIn 的 API 方法。这些方法大体上可以分成四类：

- 读取单个值，一次一个。
- 读取行，一次一行。
- 读取字符，一次一个。
- 读取相同类型的一系列值。

132

通常来说，最好不要在同一程序中混合使用不同类别的函数。以下方法中的大多数其名称即描述了它们的功能，但其精确功能还需要仔细斟酌。

public class StdIn

从标准输入读取单个数据的方法

boolean isEmpty()	标准输入是否为空（或仅有空白字符）？
int readInt()	读取一个数据，将其转换为 int 类型，然后返回
double readDouble()	读取一个数据，将其转换为 double 类型，然后返回
boolean readBoolean()	读取一个数据，将其转换为 boolean 类型，然后返回
String readString()	读取数据并将其作为字符串返回

从标准输入读取字符的方法

boolean hasNextChar()	标准输入中是否还有字符尚未读取？
char readChar()	从标准输入读取一个字符并返回

从标准输入读取行的方法

boolean hasNextLine()	标准输入中是否有下一行尚未读取？
String readLine()	读取行的其余部分并将其作为字符串返回

读取其余标准输入的方法

int[] readAllInts()	读取所有剩余的数据并将其作为 int 数组返回
double[] readAllDoubles()	读取所有剩余的数据并将其作为 double 数组返回
boolean[] readAllBooleans()	读取所有剩余的数据并将其作为 boolean 数组返回
String[] readAllStrings()	读取所有剩余的数据并将其作为 String 数组返回
String[] readAllLines()	读取所有剩余的行并将其作为 String 数组返回
String readAll()	读取其余的输入并将它作为一个字符串返回

注 1：数据是非空白字符的最大序列。

注 2：在读取数据之前，任何前导空格都将被丢弃。

注 3：有类似的方法来读取 byte、short、long 和 float 类型的值。

注 4：读取输入的每个方法如果无法读取下一个值，将抛出运行时异常，可能因为没有更多的输入或因为输入与预期类型不匹配。

133

我们的库函数 API 中提供的用于标准输入的静态方法

键入输入。当你使用 java 命令从命令行调用 Java 程序时，你实际上在做三件事情：

①发出一个命令来开始执行程序，②指定命令行参数，③开始定义标准输入流。在终端窗口中命令行之后续入的字符串即标准输入流。程序等待你在终端窗口中键入字符，当你键入字符时，你正在与你的程序进行交互。

例如，我们想要设计一个程序 `AddInts`，它首先从标准输入读入一个参数 `n`，然后从标准输入读取 `n` 个数字，并计算累加和，然后将所得的值作为标准输出。当键入“`java AddInts 4`”来运行程序时，程序会接收到命令行参数 `4`，然后调用方法 `StdIn.readInt()`，并等待键入 `4` 个整数。假设你想要输入的的第一个数字是 `144`，你键入 `1`，然后键入 `4`，再键入 `4`，但是在这之后程序没有任何反应，因为 `StdIn` 不知道你已经完成了整数的输入。你需要输入 `<Return>`（回车键——译者注）来表示整数的结尾，当按下回车键后，`StdIn.readInt()` 会立即返回 `144`，你的程序将输入累加到 `sum` 中，然后再次调用 `StdIn.readInt()`。键入第二个值之前，不会发生任何事情：如果键入 `2`，键入 `3`，键入 `3`，然后键入 `<Return>` 结束输入数字，`StdIn.readInt()` 返回 `233`，你的程序再次将输入加到 `sum`。以这种方式键入了四个数字后，`AddInts` 不再需要输入，并根据需要打印总和。

```
public class AddInts
{
    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        int sum = 0;
        for (int i = 0; i < n; i++) 解析命令行参数
        {
            int value = StdIn.readInt();
            sum += value;
        }
        StdOut.println("Sum is " + sum);
    }
}
```

从标准输入流中读取值

打印到标准输出流

命令行 命令行参数

```
% java AddInts 4
```

144
233
377
1024

← 标准输入流

Sum is 1778

标准输出流

命令解析

134

输入格式。当 `StdIn.readInt()` 期望得到一个 `int` 类型时，如果你键入 `abc` 或 `12.2` 或 `true`，因与命令所需的字符类型不符，它将返回一个名为 `InputMismatchException` 的异常。每个输入的格式必须与你在 Java 程序中指定常量时使用的格式精确匹配。为方便起见，`StdIn` 将连续空格字符的字符串视为一个空格，并允许使用空格符对数字进行分隔。格式上，在数字之间放置多个空格，或者在同一行输入数字并使用制表符分隔它们，再或者将它们分散在多行，都是允许的（除非你的终端应用程序设计为一次处理一行标准输入，在这种情况下它会等你输入 `<Return>`，然后将该行上的所有数字发送到标准输入）。你可以在输入流中混合不同类型的值，但是每当程序期望特定类型的值时，输入流必须具有该类型的值。

交互式用户输入。`TwentyQuestions`（程序 1.5.2）是一个与用户进行交互的简单示例程序。该程序生成随机整数，然后给用户线索以帮助用户猜测到该数字（通过使用二分搜索，你总是可以在 20 次问答内得到答案，参见 4.2 节）。该程序与我们编写的其他程序之间的根本区别在于：在执行程序时，用户可以更改控制流程。这一特点在早期的计算应用中是非常重要的，但是我们现在很少写这样的程序，因为现代应用程序通常会通过图形用户界面进行这样的输入，如第 3 章所讨论的。即使像 `TwentyQuestions` 这样的简单程序也说明了编写支持用户交互的程序可能非常困难，因为你必须计划好所有可能的用户输入。

135

程序1.5.2 交互式用户输入

```

public class TwentyQuestions
{
    public static void main(String[] args)
    { // 当用户尝试猜测数字时
      // 生成一个数字并回答问题
      int secret = 1 + (int) (Math.random() * 1000000);
      StdOut.print("I'm thinking of a number ");
      StdOut.println("between 1 and 1,000,000");
      int guess = 0;
      while (guess != secret)
      { // 检测一个猜测值并提供一个答案
        StdOut.print("What's your guess? ");
        guess = StdIn.readInt();
        if (guess == secret) StdOut.println("You win!");
        if (guess < secret) StdOut.println("Too low ");
        if (guess > secret) StdOut.println("Too high");
      }
    }
}

```

secret	秘密值
guess	用户的猜测

这个程序是一个简单的猜谜游戏。每当你输入一个数字，都在隐晦地向程序提出问题（是这个数字吗？），程序会告诉你的值是大还是小。当你猜对时，程序会打印“You win!”（你赢了！）但必须在20个问题以内完成。要使用此程序，必须保证StdIn和StdOut在你的Java环境中是可用的（请参阅本节末尾的第一个问答环节）。

```

% java TwentyQuestions
I'm thinking of a number between 1 and 1,000,000
What's your guess? 500000
Too high
What's your guess? 250000
Too low
What's your guess? 375000
Too high
What's your guess? 312500
Too high
What's your guess? 300500
Too low
...

```

136

处理任意大小的输入流。通常，输入流是有限的：你的程序顺序读取输入流，消耗输入流中的数据，当流为空时终止。但实际上输入流的大小没有任何限制，只是需要程序可以处理提交给它们的所有输入。程序 Average（程序 1.5.3）可以从标准输入读取一系列浮点数并打印其平均值。该程序展示了输入流的一个重要特性：对于程序来说，流的长度是未知的。我们输入所有数字，然后由程序计算它们的均值。在读取每个数字之前，程序使用 StdIn.isEmpty() 方法来检查输入流中是否还有数字。如何表示我们不再输入数据？按照惯例，我们在结束时键入一个特殊的字符序列，称为结尾符（end-of-file）。结尾符对于终端应用程序而言至关重要，不幸的是，现代操作系统中各个系统对结尾符都有各自的定义。在本书中，我们使用 <Ctrl-D> 作为结尾符（许多系统需要 <Ctrl-D> 独占一行）；还有一种广泛使用的约定是在单独的一行中键入 <Ctrl-Z>。Average 是一个简单的程序，但它展示了程序的一个重要功能：通过标准输入，我们可以编写运行程序来处理无限量的数据。显而易见，此类程序在数据处理应用中非常重要。

如程序 TwentyQuestions 和 Average 所示，标准输入是我们使用的命令行参数模型的必要步骤，原因有二：首先，通过命令行参数我们才可以与程序交互，在程序开始执行前，我们就需要向程序提供数据。其次，通过命令行参数我们才可以读取大量数据，而且只能是符

合命令行要求的值。实际上，正如程序 Average 所示，数据量可能是无限的，这个假设使得程序变得简单。标准输入的第三个要点是操作系统可以改变标准输入的来源，所以你不必从键盘键入所有的输入。接下来，我们讨论如何实现这一功能。

137

程序1.5.3 求数字流的均值

```
public class Average
{
    public static void main(String[] args)
    { // 求标准输入的均值
        double sum = 0.0;
        int n = 0;
        while (!StdIn.isEmpty())
        { // 从标准输入中读取一个数字并累计总和
            double value = StdIn.readDouble();
            sum += value;
            n++;
        }
        double average = sum / n;
        StdOut.println("Average is " + average);
    }
}
```

n	所读的数字个数
sum	累计和

该程序从标准输入读取一系列浮点数，并将其平均值打印到标准输出上（前提是总和没有溢出）。从它的角度来看，输入流的大小没有限制。下面的命令使用重定向和管道（在下一小节中讨论），提供100 000个数字求均值

```
% java Average
10.0 5.0 6.0
3.0
7.0 32.0
<Ctrl-D>
Average is 10.5
```

```
% java RandomSeq 100000 > data.txt
% java Average < data.txt
Average is 0.5010473676174824
% java RandomSeq 100000 | java Average
Average is 0.5000499417963857
```

138

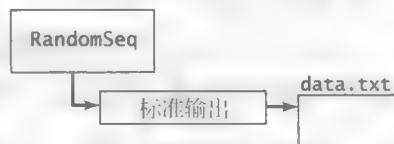
重定向和管道 对于许多应用，因为受到我们键入的数据量（以及键入速度）的限制，程序的处理能力被低估，从而使标准输入流往往会被忽视。类似地，我们经常希望保存打印在标准输出流上的信息以供后续使用。为突破这些限制，我们必须牢记：标准输入是一个抽象的概念——程序期望从输入流读取数据，但它不依赖于该输入流的来源。标准输出亦是如此。其抽象程度取决于我们（通过操作系统）为标准输入和标准输出指定各种其他来源的能力，如文件、网络程序。所有现代操作系统都实现了这种重新定义数据来源的机制。

将标准输出重定向到文件。通过在调用程序的命令中添加一个简单的指令，我们可以将其标准输出流重定向到文件中存储下来或稍后输入另一个程序。例如，

```
% java RandomSeq 1000 > data.txt
```

该行命令表示不要将标准输出流打印在终端窗口中，而是写入名为 data.txt 的文本文件中。每次调用 System.out.print() 或 System.out.println() 都会在该文件末尾附加文本。在这个例子中，最终的结果是包含 1000 个随机值的文件，而终端窗口中不显示任何输出：它被直接输出到符号“>”后指定的那个文件中。这样，我们可以保存信息以供日后检索。请注意，为实现以上功能，我们不必以任何方式更改 RandomSeq（程序 1.5.1）——这也

```
% java RandomSeq 1000 > data.txt
```



将标准输出重定向到一个文件

体现了标准输出的抽象性，也就是说，我们对这些抽象的接口给出不同形式的实现，并不影响抽象接口的使用。使用重定向可以保存任一程序输出。如果你花费了大量的努力来获得程序运行结果，则通常需要保存结果以供日后参考。在现代系统中，你可以使用剪切和粘贴或操作系统提供的类似机制来保存一些信息，但剪切和粘贴对于大量数据是不方便的。相比之下，重定向是为处理大量数据而专门设计的。

程序1.5.4 一个简单的过滤器

```
public class RangeFilter
{
    public static void main(String[] args)
    { // 过滤不在lo和hi之间的数字
        int lo = Integer.parseInt(args[0]);
        int hi = Integer.parseInt(args[1]);
        while (!StdIn.isEmpty())
        { // 处理一个数字
            int value = StdIn.readInt();
            if (value >= lo && value <= hi)
                StdOut.print(value + " ");
        }
        StdOut.println();
    }
}
```

lo	范围的下限
hi	范围的上限
value	当前数字

这个过滤器将命令行参数指定范围内的输入流中的数字复制到输出流中，流的长度没有限制

```
% java RangeFilter 100 400
358 1330 55 165 689 1014 3066 387 575 843 203 48 292 877 65 998
358 165 387 203 292
<Ctrl-D>
```

将文件重定向到标准输入。同样，我们可以重定向标准输入流，使 StdIn 从文件读取数据而不是从终端窗口读取：

```
% java Average < data.txt
```

该命令从文件 data.txt 读取一系列数字，并计算它们的平均值。具体来说，“<”符号是指示操作系统通过从文本文件 data.txt 中读取数据来实现标准输入流，而不是等待用户在终端窗口中输入内容。当程序调用 StdIn.readDouble() 时，操作系统从文件中读取值。文件 data.txt 可能由任何应用程序创建，而不仅仅是 Java 程序——你的计算机上的许多应用程序都可以创建文本文件。从文件重定向到标准输入的能力使我们能够创建数据驱动的代码（data-driven code），你可以更改程序处理的数据，而无须更改程序。同时，你可以在文件中保存数据并编写一个从标准输入流读取的程序。



将文件重定向到标准输入

连接两个程序。实现标准输入和标准输出最灵活的方式是由我们自己的程序实现它们。这种机制称为管道（pipng）。例如，命令

```
% java RandomSeq 1000 | java Average
```

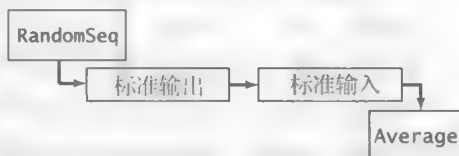
规定了 RandomSeq 的标准输出流和 Average 的标准输入流是相同的流。它的效果是在

Average 运行时，RandomSeq 会将其生成的数字输入到终端窗口。该示例也具有与以下命令相同的效果：

```
% java RandomSeq 1000 > data.txt
% java Average < data.txt
```

只是在这种情况下不必创建 data.txt。这种区别有非常重要的意义，因为它消除了我们可以处理的输入输出流大小的另一个限制。例如，即使你的计算机上可能没有空间保存十亿个数字（但是，你需要时间来处理它们），你也可以在示例中将 1000 替换为 1000000000。当 RandomSeq 调用 System.out.println() 时，添加一个字符串到流的末尾；当 Average 调用 StdIn.readInt() 时，从流的开始处删除一个字符串。具体先调用哪个取决于操作系统：它可能会运行 RandomSeq，直到它产生一些输出，然后运行 Average 来消耗该输出，或者它可能运行 Average 直到需要一些输入，然后运行 RandomSeq，直到它产生所需的输入。最终的结果是一样的，但是你的程序不必担心这些细节，它们仅与标准输入和标准输出抽象一起工作。

```
% java RandomSeq 1000 | java Average
```



将一个程序的输出输送到另一个程序的输入

141

过滤器。管道是 20 世纪 70 年代初 UNIX 系统的核心特性功能之一，并仍保留在现代系统中，因为它能够将不同程序之间进行的通信简单化。例如，许多 UNIX 程序现在仍在使用，并且要处理比程序作者当初设想的大成千上万倍的文件。通过标准输入和标准输出，我们可以用方法调用其他 Java 程序，也可以调用之前编写的程序，或者用另一种语言编写的程序。通过标准输入和标准输出，我们与外部世界建立了一个简单的接口。

常见的例子程序之一就是过滤器：其将一个标准输入流以某种方式转换为标准输出流，以管道作为纽带将程序连接在一起。例如，RangeFilter（程序 1.5.4）需要两个命令行参数，并在标准输出上打印来自标准输入指定范围的数字。你可以将标准输入设想为来自某些仪器的测量数据，并使用过滤器将实验不感兴趣的数据丢弃。

早期为 UNIX 设计的几个标准过滤器现在仍在使用（其中有一些更换了名称），有些甚至已经成为现代操作系统中的命令。例如，过滤器 sort 能够将标准输入上传来的数据按照从小到大的顺序放到标准输出上。

```
% java RandomSeq 6 | sort
0.035813305516568916
0.14306638757584322
0.348292877655532103
0.5761644592016527
0.7234592733392126
0.9795908813988247
```

我们将在 4.2 节讨论排序。第二个有用的过滤器是 grep，它从标准输入打印匹配给定模式的行。例如，如果你键入：

```
% grep lo < RangeFilter.java
```

会得到如下结果：

```
// 过滤不在 lo 和 hi 之间的数字
int lo = Integer.parseInt(args[0]);
if (value >= lo && value <= hi)
```

142

程序员经常使用诸如 `grep` 等工具来快速找到变量名或语言的使用细节信息。第三个有用的过滤器是 `more`，它从标准输入读取数据，并将其显示在终端窗口中，每次显示整个屏幕。例如，如果你输入

```
% java RandomSeq 1000 | more
```

你将在终端窗口中看到满屏数字，但是会有更多的数字等待你点击空格键后显示在下一屏中。术语过滤器（filter）可能会有一些误导：它的原意是描述像 `RangeFilter` 这样的程序，`RangeFilter` 将标准输入的一些子序列写入标准输出，但是现在过滤器常用于描述从标准输入读取并写入标准输出的程序。

多个流。对于许多常见程序任务，我们希望能够从多个来源获取输入，或者为多个目标产生输出。在 3.1 节中，我们讨论了 `Out` 库和 `In` 库，它们扩展了 `StdOut` 和 `StdIn` 以允许多个输入输出流。这些库所重定向的数据流不仅可以来自文件，还能来自网页。

处理大量信息在许多计算应用中起着至关重要的作用。科学家可能需要分析从一系列实验中收集的数据，股票交易者可能希望分析关于最近金融交易的信息，学生可能希望保存他们收藏的音乐和电影。在这些以及很多其他应用中，数据驱动程序成为规范，而通过使用标准输出、标准输入、重定向和管道，我们可以在 Java 程序中实现这些应用和功能。我们可以通过网络或任何标准设备将数据收集到计算机的文件中，并使用重定向和管道将数据连接到我们的程序。本书中的许多编程示例都具有这种能力。

[143]

标准绘图 到目前为止，我们的输入/输出仅专注于文本字符串。现在我们介绍如何生成图像来作为输出。这个库很容易使用，并且允许我们利用视觉媒介来传递信息，这通常比文本具有更强大的表现力。

就像 `StdIn` 和 `StdOut`，我们的标准绘图在库 `StdDraw` 中实现，你需要将相应的文件配置在 Java 环境中（请参阅本节结尾的第一个问答环节）。标准绘图非常简单。我们设计出一个抽象的绘图设备，能够在二维画布上绘制线和点，该设备能够通过 `StdDraw` 中的方法调用形式发出命令，如下所示：

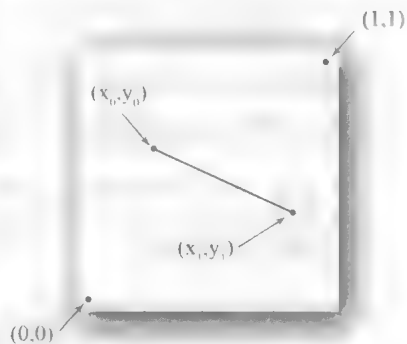
```
public class StdDraw (基础绘图命令)
```

```
void line(double x0, double y0, double x1, double y1)
```

```
void point(double x, double y)
```

与标准输入和标准输出的方法一样，这些方法的名称基本上可以表示它的功能。例如：`StdDraw.line()` 用于绘制一条将点 (x_0, y_0) 与点 (x_1, y_1) 连接起来的直线段，坐标由参数给出；`StdDraw.point()` 以参数给出的坐标 (x, y) 为中心绘制一个点，其大小为默认的单位刻度（所有 x 坐标和 y 坐标位于 0 和 1 之间）。`StdDraw` 以计算机屏幕的窗口为画布，背景为白色，线和点为黑色。该窗口有一个菜单选项，可以将绘图保存到文件中，并可以打印在纸上或发布到网上。

你的第一幅画。类似于编写程序的第一步是从 `Hello-World` 开始，使用 `StdDraw` 编程图形的第一步是 `Triangle`，即绘制一个内部包含一个点的等边三角形。为了构成三角形，我们绘制三条线段：一条从左下角的点 $(0,0)$ 到点



[144]

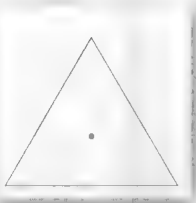
(1,0)，一条从点 $(1,0)$ 到点 $(\frac{1}{2}, \frac{\sqrt{3}}{2})$ ，还有一条从点 $(\frac{1}{2}, \frac{\sqrt{3}}{2})$ ，`StdDraw.line(x0, y0, x1, y1);`

$\frac{\sqrt{3}}{2}$) 到 (0,0)。最后一步, 我们在三角形中间画一个点。一旦你成功编译并运行了 Triangle, 你就可以编写自己的程序来绘制由线段和点组成的图形。这项能力会使你的输出更加丰富。

当你使用计算机创建图像时, 你会立即获得反馈 (图像), 以便快速改进你的程序。通过计算机程序, 你可以创建依靠手工制作无法完成的图像。特别是我们可以使用更加具有表现力的图片来展示数据, 而不是仅仅将数据视为数字。在讨论其他绘图命令之后, 我们将研究其他图形示例。

控制命令。默认画布的尺寸是 512×512 像素; 如果要更改它, 请在使用任何绘图命令之前调用 setCanvasSize()。标准绘图的默认坐标系是单位正方形, 但我们经常想绘制不同尺度的图像。例如, 常见的情况是我们希望把 x 坐标或者 y 坐标设定在某些范围内, 有时可能需要同时设定 x 坐标和 y 坐标的范围。此外, 我们经常想要依据标准绘制不同粗细的线段和不同尺寸的点。为了满足这些需求, StdDraw 具有以下方法:

```
public class Triangle
{
    public static void main(String[] args)
    {
        double t = Math.sqrt(3.0)/2.0;
        StdDraw.line(0.0, 0.0, 1.0, 0.0);
        StdDraw.line(1.0, 0.0, 0.5, t);
        StdDraw.line(0.5, t, 0.0, 0.0);
        StdDraw.point(0.5, t/3.0);
    }
}
```



你的第一幅画

public class StdDraw (基础控制命令)

void setCanvasSize(int w, int h)	在屏幕上创建一个宽为w和高为h (以像素为单位) 的画布
void setXscale(double x0, double x1)	将x-scale重置为 (x0, x1)
void setYscale(double y0, double y1)	将y-scale重置为 (y0, y1)
void setPenRadius(double radius)	将画笔的半径设置为radius

注意: 具有相同名称但无参数的方法重置为默认值, x-scale和y-scale的默认值是单位平方, 画笔半径的默认值是0.002

145

例如, 两个调用序列:

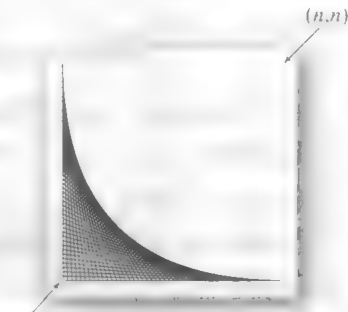
```
StdDraw.setXscale(x0, x1);
StdDraw.setYscale(y0, y1);
```

将绘图坐标设置在左下角位于 (x₀, y₀) 并且右上角位于 (x₁, y₁) 的边界框 (bounding box) 内。缩放是图形中常用的简单变换。在本章的应用中, 我们直接使用它来将图形与数据相匹配。

笔是圆形的, 所以线有圆形的端点, 当你将笔的半径设为 r 并绘制一个点时, 会得到半径为 r 的圆。默认笔半径为 0.002, 不受坐标缩放的影响。这个默认值约为默认窗口宽度的 1/500, 因此如果沿水平或垂直线绘制等间距的 200 个点, 你将可以看到各个独立的圆圈, 但是如果绘制 250 个这样的点, 看起来则是像一条线。当使用 StdDraw.setPenRadius (0.01) 命令时, 表示线段的厚度和点的大小是 0.002 标准的 5 倍。

将数据过滤到标准绘图。标准绘图最简单的应用之一是将标准输入过滤到标准绘图来绘制数据图。PlotFilter (程序 1.5.5) 是一个过滤器: 它从标准输入读取由 (x, y) 坐标定义的一系

```
int n = 50;
StdDraw.setXscale(0, n);
StdDraw.setYscale(0, n);
for (int i = 0; i <= n; i++)
    StdDraw.line(0, n-i, i, 0);
```



缩放到整数坐标

列点，并在每个坐标处绘制一个点。按照约定，标准输入上的前四个数字用于指定边界框，以便它缩放图形，而不必再通过所有点来确定比例。以这种方式绘制的点的图形表示比数字本身更具表现力（而且更简洁）。相比一个坐标列表，通过程序 1.5.5 生成的图像更容易推断点的属性（例如，当用绘制的点来表示城市位置时，很容易推断出人口中心的分布趋势）。每当我们处理代表物理世界的的数据时，可视化的图像很可能是我们用来显示输出的最有意义的方法之一。PlotFilter 程序展示了如何轻松地创建这样的图像。

146

程序1.5.5 标准输入到绘图过滤器

```
public class PlotFilter
{
    public static void main(String[] args)
    {
        // 按照前四个值进行缩放
        double x0 = StdIn.readDouble();
        double y0 = StdIn.readDouble();
        double x1 = StdIn.readDouble();
        double y1 = StdIn.readDouble();
        StdDraw.setXscale(x0, x1);
        StdDraw.setYscale(y0, y1);

        // 读取点坐标并使用标准绘图画出点
        while (!StdIn.isEmpty())
        {
            double x = StdIn.readDouble();
            double y = StdIn.readDouble();
            StdDraw.point(x, y);
        }
    }
}
```

x0	左边界
y0	下边界
x1	右边界
y1	上边界
x, y	当前点

该程序从标准输入读取一系列点，并将其绘制到标准绘图（根据惯例，前四个数字是最小的x、y坐标和最大的x、y坐标）。文件USA.txt包含美国13 509个城市的坐标。

```
% java PlotFilter < USA.txt
```



147

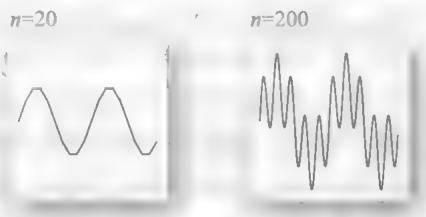
绘制函数图。标准绘图的另一个重要用途是绘制实验数据或数学函数的图像。例如，假设我们要在区间 $[0, \pi]$ 上绘制函数 $y = \sin(4x) + \sin(20x)$ 的图像。这样的任务是一个典型的采样（sample）过程：在这个区间里有无数个点，但我们必须从这些点中选取有限个点来表示这个函数。对这个函数的采样过程，需要选择一组 x ，然后计算这些 x 在函数中对应的 y 值，然后用线将这些连续的点连起来以绘制函数图，这就是分段线性近似（piecewise linear approximation）。最简单的方法是均匀地抽取 x 。首先，先确定样本大小，用区间大小除以样本大小来确定 x 的值。为了确保要绘制的点落在可视化的画布内，根据区间设置 x 轴范

围, 根据函数在这个区间的最大值和最小值设置 y 轴范围。曲线的弯曲度取决于函数的特性和样本的大小。如果样本量太小, 则函数的图像可能不太准确 (可能不太平滑, 甚至会错过主要的波动); 如果样本量太大, 则生成图像可能很耗时, 因为某些函数的计算比较耗时 (在 2.4 节中, 我们将介绍一种绘制平滑曲线而无需使用过多数量的方法)。你可以使用相同的技术绘制任何函数的函数图。也就是说, 你可以决定你要绘制的函数图像的 x 取值间隔, 计算在该间隔内的函数值, 确定并设置 y 的范围, 最后画线将这些点连接起来。

轮廓和填充形状。StdDraw 还包括绘制圆形、正方形、矩形和任意多边形的方法。每个形状都有一个轮廓。当方法名称是形状的名称时, 表示的是使用绘图笔描出该形状的轮廓。当方法的名称以 `filled` 开始时, 这个名称对应的形状会被填充为实心的。以上方法我们总结在如下 API 中:

```
double[] x = new double[n+1];
double[] y = new double[n+1];
for (int i = 0; i <= n; i++)
    x[i] = Math.PI * i / n;
for (int i = 0; i <= n; i++)
    y[i] = Math.sin(4*x[i]) + Math.sin(20*x[i]);
StdDraw.setXscale(0, Math.PI);
StdDraw.setYscale(-2.0, 2.0);
for (int i = 1; i <= n; i++)
    StdDraw.line(x[i-1], y[i-1], x[i], y[i]);
```

148

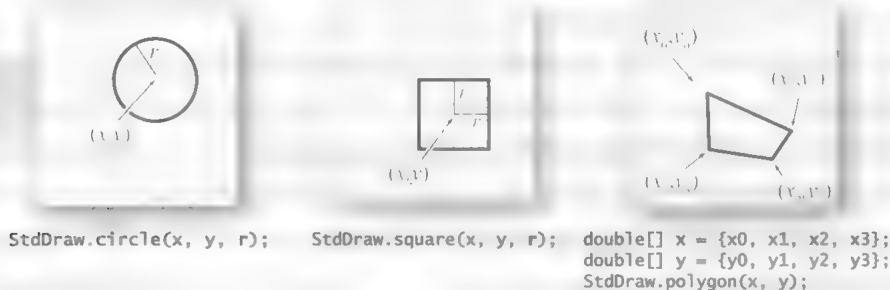


绘制一个函数图

```
public class StdDraw (形状)
```

```
void circle(double x, double y, double radius)
void filledCircle(double x, double y, double radius)
void square(double x, double y, double r)
void filledSquare(double x, double y, double r)
void rectangle(double x, double y, double r1, double r2)
void filledRectangle(double x, double y, double r1, double r2)
void polygon(double[] x, double[] y)
void filledPolygon(double[] x, double[] y)
```

`circle()` 和 `filledCircle()` 的参数定义了一个圆心为 (x, y) 、半径为 r 的圆; `square()` 和 `filledSquare()` 的参数定义了以 (x, y) 为中心、边长为 $2r$ 的正方形; `rectangle()` 和 `filledRectangle()` 的参数定义了以 (x, y) 为中心、宽 $2r$ 和高 $2s$ 的矩形; `polygon()` 和 `filledPolygon()` 的参数定义了一个由线段连接的点序列, 包括连接第一个点和最后一点的序列。



```
StdDraw.circle(x, y, r);
```

```
StdDraw.square(x, y, r);
```

```
double[] x = {x0, x1, x2, x3};
double[] y = {y0, y1, y2, y3};
StdDraw.polygon(x, y);
```

149

文本和颜色。有时你可能希望在图像中标注或突出显示各种元素。StdDraw 具有绘制文

本的方法、用于设置与文本相关参数的方法，以及用于设置画笔颜色的方法。我们在本书中很少使用这些功能，但它们非常有用，特别是对于计算机屏幕上的图像。你会在本书官网上发现许多使用它们的例子。

```
public class StdDraw ( 文本和颜色命令 )
{
    void text(double x, double y, String s)
    void setFont(Font font)
    void setPenColor(Color color)
}
```

在这段代码中，Font 和 Color 不是基本数据类型，你将在 3.1 节中了解到如何定义这些类型。在此之前，我们将细节留给 StdDraw。可用的颜色有黑色 (BLACK)、蓝色 (BLUE)、青色 (CYAN)、深灰色 (DARK_GRAY)、灰色 (GRAY)、绿色 (GREEN)、浅灰色 (LIGHT_GRAY)、品红色 (MAGENTA)、橙色 (ORANGE)、粉红色 (PINK)、红色 (RED)、白色 (WHITE)、黄色 (YELLOW) 和浅蓝色 (BOOK_BLUE)，所有这些颜色都被定义为 StdDraw 中的常量。默认颜色为黑色，假如调用 StdDraw.setPenColor (StdDraw.GRAY) 则改为灰色。StdDraw 中的默认字体适用于你需要的大部分图形（可以在本书官网上找到使用其他字体的信息）。使用这些方法函数，你可以对绘制的图像进行必要的信息标注，例如，你可能会使用高亮的方法来标注函数图以凸显相关值。

形状、颜色和文本是基本的工具，可用于绘制令人眼花缭乱的各种图像，但你应该谨慎使用它们。使用这样的工具通常会带来设计上的挑战，而且我们设计 StdDraw 虽然遵循现代图形库标准，但提供的命令是粗糙的，因此你可能需要大量的调用才能产生你设想的美丽图像。相比之下，使用颜色或标签来帮助标识图像中的重要信息则易于实现，如使用颜色来表示数据值。

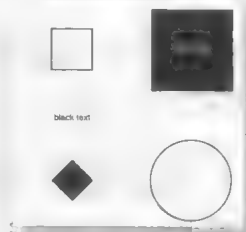
双缓冲。StdDraw 支持一种强大的计算机图形机制，称为双缓冲 (double buffering)。当通过调用 enableDoubleBuffering() 启用双缓冲时，所有绘图都将发生在缓冲区画布 (offscreen canvas) 上。缓冲区画布不显示；它只存在于计算机内存中。只有当你调用 show() 时，你的图像才从缓冲区画布复制到屏幕画布 (onscreen canvas) 上，并在标准绘图窗口中显示。你可以将双缓冲看作收集所有要绘制的线条、点、形状和文字的容器，然后根据请求同时绘制缓冲区中的内容。双缓冲使得你对绘图的控制更加精确。

使用双缓冲的优势在于执行大量绘图命令时的效率。创建复杂绘图时，其逐步生成的过程是难以想象的。例如，你可以通过在 while 循环之前调用 enableDoubleBuffering()，之后调用 show() 来加快程序 1.5.5。这会使得所有的点一次同时出现，而非每次出现一个。

双缓冲最重要的用途是制作计算机动画 (computer animations)，通过快速显示静态图形来产生运动的幻觉。这种效果可以使科学现象动态可视化。重复以下四个步骤就可产生动画：

- 清除缓冲区画布。
- 在缓冲区画布上绘制对象。

```
StdDraw.square(.2, .8, .1);
StdDraw.filledSquare(.8, .8, .2);
StdDraw.circle(.8, .2, .2);
double[] xd = { .1, .2, .3, .2 };
double[] yd = { .2, .3, .2, .1 };
StdDraw.filledPolygon(xd, yd);
StdDraw.text(.2, .5, "black text");
StdDraw.setPenColor(StdDraw.WHITE);
StdDraw.text(.8, .8, "white text");
```



形状和文本示例

- 将缓冲区画布复制到屏幕画布上。
- 稍等片刻。

为了支持第一步和最后一步，StdDraw 提供了三个附加方法。clear() 方法将画布清除为白色或指定颜色；pause() 方法控制动画的显示速度，其参数 dt 表示在处理其他命令前 StdDraw 等待 dt 毫秒。

public class StdDraw (高级控制命令)	
void enableDoubleBuffering()	启用双缓冲
void disableDoubleBuffering()	关闭双缓冲
void show()	将缓冲区画布复制到屏幕画布上
void clear()	清空画布并将画布颜色设置为白色(默认)
void clear(Color color)	清空画布并将画布颜色设置为参数color的颜色
void pause(double dt)	暂停dt毫秒

151

弹跳球。动画制作的“Hello, World”程序是制作一个在画布上移动的黑色球，球根据完全弹性碰撞规律弹离边界（即速度方向反转，速率没有损失——译者注）。假设球位于位置 (r_x, r_y) ，我们想创建一个将其移动到附近位置的效果，假设这个位置是 $(r_x+0.01, r_y+0.02)$ 。我们分四个步骤来完成这个动作：

- 将缓冲区画布清除为白色。
- 在缓冲区画布上的新位置画一个黑色球。
- 将缓冲区画布复制到屏幕画布上。
- 稍等片刻。

为了创造出运动的假象，我们对这个球的整个位置进行迭代（在这种情况下将形成一条直线）。没有双缓冲的话，球的图像将在黑色和白色之间快速闪烁，而非平滑的动画。BouncingBall（程序 1.5.6）执行这些步骤来创建一个球在以原点为中心的 2×2 盒中移动的假象。球的当前位置是 (r_x, r_y) ，每一步， r_x 加 v_x ， r_y 加 v_y ，以此来计算球的新位置。 (v_x, v_y) 是球在每个时间单位中移动的固定距离，所以它代表球的速度（velocity）。为了将球保持在标准绘图窗口中，我们根据弹性碰撞的规律模拟球从墙壁上弹起的效果。这种效果很容易实现：当球撞到垂直的墙壁时，我们将 x 方向的速度从 v_x 改变为 $-v_x$ ，当球撞到水平墙时，我们将 y 方向的速度从 v_y 改为 $-v_y$ 。当然，你需要从本书官网上下载代码并在计算机上运行代码，才能清楚地查看到球的运动状态。为了使打印页面上的图像更清晰，我们修改了 BouncingBall，使用灰色背景，也可以显示球的移动轨迹（参见练习 1.5.34）。

把标准绘图功能添加到我们的编程模型中，使我们可以“一图胜千言”。这是一套非常易于使用的编程接口，通过它，程序可以更开放地面对真实世界，也可以轻松地实现科学和工程函数和数据的可视化。我们将在本书中经常使用到这些功能。因此在最近几个示例程序上花费精力是非常值得的。在本书官网和课后练习中有更多有用的应用示例，同时你也可以深度使用 StdDraw 来应对各种挑战及实现自我创意，例如，如何画一个 n 角星？如何让反弹球根据实际情况反弹（增加重力）？你会惊奇地发现实现这些或其他任务是多么的容易。

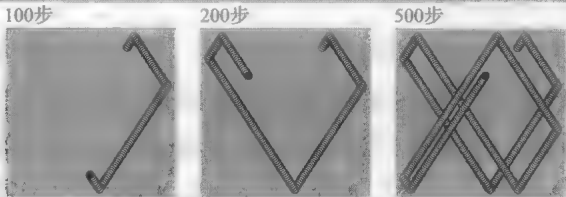
152

程序1.5.6 弹跳球

```
public class BouncingBall
{
    public static void main(String[] args)
    { // 模拟弹跳球的运动
        StdDraw.setXscale(-1.0, 1.0);
        StdDraw.setYscale(-1.0, 1.0);
        double rx = 0.480, ry = 0.860;
        double vx = 0.015, vy = 0.023;
        double radius = 0.05;
        StdDraw.enableDoubleBuffering();
        while(true)
        { // 更新球位置并绘制
            if (Math.abs(rx + vx) + radius > 1.0) vx = -vx;
            if (Math.abs(ry + vy) + radius > 1.0) vy = -vy;
            rx += vx;
            ry += vy;
            StdDraw.clear();
            StdDraw.filledCircle(rx, ry, radius);
            StdDraw.show();
            StdDraw.pause(20);
        }
    }
}
```

rx, ry	位置
vx, vy	速度
dt	等待时间
radius	球半径

该程序模拟弹跳球在坐标为-1和+1之间的框中的运动。球根据弹性碰撞规律从边界反弹。即使球的大部分像素在黑色和白色之间交替，StdDraw.pause()用20毫秒的等待时间保持黑色图像的球持续出现在屏幕上。下图显示球的轨迹由该代码的修改版本生成（参见练习1.5.34）。



153

以下 API 表总结了我们的研究过的 StdDraw 方法：

public class StdDraw	
绘图命令	
void line(double x0, double y0, double x1, double y1)	
void point(double x, double y)	
void circle(double x, double y, double radius)	
void filledCircle(double x, double y, double radius)	
void square(double x, double y, double radius)	
void filledSquare(double x, double y, double radius)	
void rectangle(double x, double y, double r1, double r2)	
void filledRectangle(double x, double y, double r1, double r2)	
void polygon(double[] x, double[] y)	
void filledPolygon(double[] x, double[] y)	
void text(double x, double y, String s)	
控制命令	
void setXscale(double x0, double x1)	将x坐标重置为 (x0,x1)

<code>void setYscale(double y0, double y1)</code>	将y坐标重置为 (y0,y1)
<code>void setPenRadius(double radius)</code>	将笔半径设置为radius
<code>void setPenColor(Color color)</code>	设置笔的颜色为color
<code>void setFont(Font font)</code>	设置文本字体为font
<code>void setCanvasSize(int w, int h)</code>	将画布大小设置为宽w, 高h
<code>void enableDoubleBuffering()</code>	启用双缓冲
<code>void disableDoubleBuffering()</code>	关闭双缓冲
<code>void show()</code>	将缓冲区画布复制到屏幕画布
<code>void clear(Color color)</code>	清空画布并将画布颜色设置为color
<code>void pause(int dt)</code>	暂停dt毫秒
<code>void save(String filename)</code>	保存为.jpg文件或.png文件

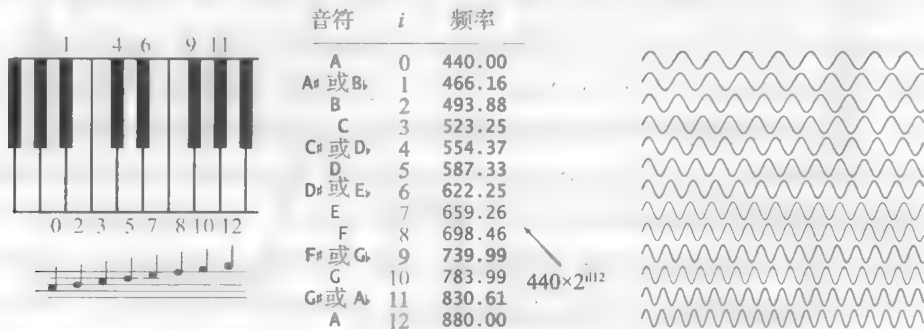
注意：具有相同名称但无参数的方法重置为默认值。

标准绘图静态方法库的 API

154

标准音频 作为输出的基本抽象的最后一个示例，我们来研究 StdAudio，一个可用于播放、处理和合成声音的库。你可能已经用计算机处理过音乐，现在你可以通过编写程序来做这些事。同时，你还将学习计算机科学与科学计算领域的一个重要概念——数字信号处理 (digital signal processing)。当揭开了这个学科的迷人面纱时，你可能会惊讶于它基本概念的简洁。

Concert A。 声音是对分子振动的感知——特别是我们鼓膜的振动。因此，振动是理解声音的关键。也许最简单的入门方式是研究高于中央 C 的音符 A，称为 Concert A。这个音符就是一个正弦波，以每秒 440 次的频率振荡。函数 $\sin(t)$ 的周期是 2π 个单位，所以如果我们以秒为单位测量 t 并绘制函数 $\sin(2\pi t \times 440)$ ，我们得到一个每秒振荡 440 次的曲线。当你通过拨吉他琴弦、吹小号或使扬声器中的小锥体振动来演奏 A 时，你听到的主要就是**这个正弦波，也就是我们说的 concert A 声音**。我们以赫兹（每秒周期数）为单位测量频率。当你将频率加倍或减半时，你可以在刻度上向上或向下移动一个八度音。例如，880 赫兹比 concert A 高一个八度音，110 赫兹比 Concert A 低两个八度音。作为参考，人类听力的频率范围约为 20 至 20 000 赫兹。声音的幅度 (y 值) 对应音量。我们绘制值域在 -1 和 +1 之间的曲线，并假设任何录制和播放声音的设备都会根据需要进行缩放，当转动音量旋钮时，将进一步缩放。



音符、数字和波形

155

其他音符。 一个简单的数学公式表征了半音音阶上的其他音符。半音音阶上有 12 个音符，以对数（底数为 2）刻度均匀间隔。我们将它的频率乘以 2 的 $(i/12)$ 次方得到给定音

符上方的第 i 个音符。换句话说，半音音阶中的每个音符的频率正好是音阶中前一音符的频率乘以 2 的 $1/12$ 次方（约 1.06）。你会惊喜地发现，有了这些信息就可以用来创造音乐了！例如，要播放曲子《Frère Jacques》，只需按照适当频率产生正弦波来演奏音符“A B C # A”，每个音符半秒，然后重复。StdAudio 库中的基础方法 StdAudio.play() 可以完成此操作。

采样。对于数字声音，我们以规定的间隔进行采样来表示曲线，与我们绘制函数图时的方式完全相同。我们经常通过充分采样以准确地表示曲线——广泛使用的数字声音采样率是每秒 44 100 个样本。对于 Concert A，该采样率相当于绘制正弦波的每个周期需要大约 100 个样本。由于我们使用标准化的采样时间间隔，我们只需要计算采样点的 y 坐标。这很简单：我们将声音表示为实数数组（在 -1 和 +1 之间）。按照这个设计方法，StdAudio.play() 方法需要一个数组作为参数，数组中的每一项表示的是相应采样的音量，运行该方法能够在计算机上演奏该数组表示的音乐。

例如，假设你想播放 Concert A 10 秒钟，每秒 44 100 个样本，你需要一个长度为 441 001 的 double 类型数组。为了填充数组，使用 for 循环在 $t=0/44\ 100$ 、 $1/44\ 100$ 、 $2/44\ 100$ 、 $3/44\ 100$ 、 \dots 、 $441\ 000/44\ 100$ 处对函数 $\sin(2\pi t \times 440)$ 进行采样。一旦我们得到数组的各项值，我们就可以使用 StdAudio.play() 了。代码如下：

```
int SAMPLING_RATE = 44100;      // 每秒样本数
int hz = 440;                   // concert A
double duration = 10.0;         // 10秒
int n = (int) (SAMPLING_RATE * duration);
double[] a = new double[n+1];
for (int i = 0; i <= n; i++)
    a[i] = Math.sin(2*Math.PI * i * hz / SAMPLING_RATE);
StdAudio.play(a);
```

156

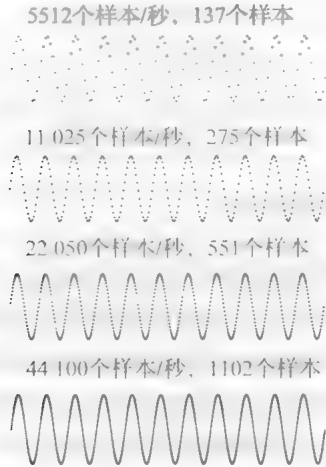
这段代码是数字音频的“Hello, World”。一旦你理解如何使用它来让你的计算机播放这个音符，你可以编写代码来播放其他音符和制作音乐！创建声音和绘制振荡曲线之间的区别不过是输出设备不同。实际上，向标准绘图和标准音频发送相同的数字是非常有趣的，也会给你带来很多启发（见练习 1.5.27）。

保存到文件。音乐须占用计算机的大量存储空间。每秒 44 100 个样本，四分钟的歌曲对应于 $4 \times 60 \times 44\ 100 = 10\ 584\ 000$ 个数字。因此，通常使用二进制格式来表示对应歌曲的数字，这样占用的空间要比我们用字符数字表示占用的空间要小。近年来开发出了许多这样的格式——StdAudio 使用 .wav 格式。你可以在本书官网上找到关于 .wav 格式的一些信息，但是你不需要了解详细信息，因为 StdAudio 会为你处理转换。我们的标准音频库允许你读取 .wav 文件、写入 .wav 文件、将 .wav 文件转换为 double 类型数组以进行处理。

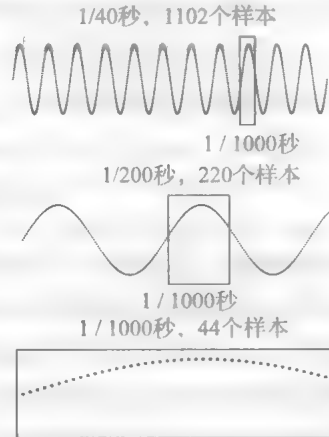
PlayThatTune（程序 1.5.7）是一个示例，显示如何使用 StdAudio 将你的计算机变成乐器。它从标准输入得到音符，从 Concert A 的半音音阶上索引，并在标准音频播放。这只是一个基本方案，你可以想到很多方式扩展其功能，我们在课后练习中也做了一些扩展。

之所以能够在基本的编程工具库中引入标准音频，是因为声音的处理是一种非常重要的科学计算的应用。从应用的角度看，数字信号处理的商业化应用对现代社会产生了显著的影响，而从科学和工程的角度看，它是物理学与计算机科学的有趣结合。本书稍后将详细研究数字信号处理的更多内容（例如，你将在 2.1 节中学习如何创建比 PlayThatTune 产生的纯音乐更具乐感的声音）。

1/40秒（各种采样率）



44 100个样本/秒（各种时间）



对正弦波采样

157

程序1.5.7 数字信号处理

```
public class PlayThatTune
{
    public static void main(String[] args)
    { // 从StdIn读一首曲子并播放它
        int SAMPLING_RATE = 44100;
        while (!StdIn.isEmpty())
        { // 读取一个音符并播放它
            int pitch = StdIn.readInt();
            double duration = StdIn.readDouble();
            double hz = 440 * Math.pow(2, pitch / 12.0);
            int n = (int) (SAMPLING_RATE * duration);
            double[] a = new double[n+1];
            for (int i = 0; i <= n; i++)
                a[i] = Math.sin(2*Math.PI * i * hz / SAMPLING_RATE);
            StdAudio.play(a);
        }
    }
}
```

pitch	到A的距离
duration	音符播放时间
hz	频率
n	样本数
a[]	采样后的正弦波

该数据驱动程序将你的计算机变成乐器。它从标准输入读取音符和持续时间，并按照规定时间播放音符代表的纯音。每个音符被指定为一个音高（与Concert A的距离）。读取每个音符和持续时间后，该程序以每秒44 100个样本的指定采样频率和持续时间对正弦波采样来创建数组，并使用StdAudio.play()播放它

```
% more elise.txt
7 0.25
6 0.25
7 0.25
6 0.25
7 0.25
2 0.25
5 0.25
3 0.25
0 0.50
```



```
% java PlayThatTune < elise.txt
```

158

下面的 API 表总结了 StdAudio 中的方法：

```

public class StdAudio
{
    void play(String filename)           播放给定的.wav文件
    void play(double[] a)               播放给定的声波
    void play(double x)                 将x作为采样样本播放1/44 100秒
    void save(String filename, double[] a) 保存到一个.wav文件
    double[] read(String filename)       从一个.wav文件读取
}

```

标准音频静态方法库的 API

总结 标准输入、标准输出、标准绘图和标准音频抽象化的益处尽可以淋漓尽致地体现在 I/O 上：它可以在不同时间连接到不同的物理设备，并且不必对程序进行任何更改。虽然设备有显著差异，但我们可以编写能够进行 I/O 的程序，而不依赖于特定设备的属性。从现在开始，我们将在本书的几乎所有程序中使用 StdOut、StdIn、StdDraw 和 / 或 StdAudio 的方法。为了方便，我们将这些库统称为 Std*。使用这些库的重要优势在于可以将程序切换到更快、更廉价或能够存储更多数据的新设备上，而无须更改程序。在这种情况下，连接的细节是你的操作系统和 Std * 实现之间需要解决的问题。在现代系统中，新设备通常配有相应的软件，能够自动地与 Java 和操作系统建立连接。

159

问答环节

问：怎样将 StdIn、StdOut、StdDraw 和 StdAudio 用于 Java？

答：如果你按照本书官网中的说明一步步安装 Java，则这些库应该已经能在 Java 中使用了。或者，你可以从本书官网中复制 StdIn.java、StdOut.java、StdDraw.java 和 StdAudio.java 文件，并与使用它们的程序放在同一目录下。

问：错误消息异常 “Exception in thread “main” java.lang.NoClassDefFoundError: StdIn” 是什么意思？

答：库 StdIn 没有配置到你的 Java 环境中，请参考上一题进行配置。

问：为什么我们不使用标准 Java 库处理输入、图形和声音？

答：我们正在使用它们，但我们喜欢用更简单的抽象模型。StdIn、StdDraw 和 StdAudio 背后的 Java 库用于产品编程，这些库和它们的 API 不够轻便。要了解它们是什么样子，请查看 StdIn.java、StdDraw.java 和 StdAudio.java 中的代码。

问：如果我使用 %2.4f 这种格式来表示 double 类型值，我将得到一个 小数点前有 2 位、小数点后有 4 位的数字，对吗？

答：不，它表示小数点后有 4 位数字。第一个值是 整个字段的宽度。你要使用格式 %7.2f 指定一共有 7 个字符，小数点前有 4 位，小数点占一位，小数点后有 2 位。

问：printf() 有哪些其他转换码？

答：对于整数值，o 表示八进制，x 表示十六进制。日期和时间也有许多格式。有关详细信息请参阅本书官网。

问：我的程序可以从标准输入重新读取数据吗？

160

答：不可以，你只能读取一次，就像你无法撤销 println() 命令一样。

问：如果我的程序在耗尽标准输入中的值后尝试从标准输入读取数据，会发生什么？

答：你会得到一个错误。StdIn.isEmpty() 允许你通过检查是否还有更多的输入，以此来避免这样的错误。

问：为什么 `StdDraw.square(x, y, r)` 绘制的正方形的宽度为 $2r$ 而不是 r ？

答：这使得它与函数 `StdDraw.circle(x, y, r)` 一致，其中第三个参数是圆的半径，而不是直径。在这个例子中， r 是能够在正方形内部画出的最大圆的半径。

问：我的终端窗口在 `StdAudio` 程序结束时挂起，我不得不使用 `<Ctrl-C>` 让命令提示符重新显示出来。如何避免这种现象？

答：添加一个调用 `System.exit(0)` 作为 `main()` 的最后一行。不要问为什么。

问：为 `PlayThatTune` 制作输入文件时，能否使用负整数来指定低于 `Concert A` 的音符？

答：可以。其实我们把 `Concert A` 置于 0 是随意选的，并不是一个固定的设计。一个流行的标准称为 MIDI 调音标准，它从低于 `Concert A` 五个八度的 C 开始编号。根据 MIDI 标准，`Concert A` 为 69，你不需要使用负数。

问：当我尝试校验 30 000 赫兹（或更高）频率的正弦波时，为什么在标准音频上听到奇怪的结果？

答：奈奎斯特频率（Nyquist frequency）定义为采样频率的一半，表示可以再现的最高频率。对于标准音频，采样频率为 44 100 赫兹，奈奎斯特频率为 22 050 赫兹。

161

练习

- 1.5.1 编写从标准输入读取整数（用户输入多少读取多少）的程序，并打印最大值和最小值。
- 1.5.2 修改前一个练习的程序，确保整数必须是正的（当输入的值不是正数时提示用户输入正整数）。
- 1.5.3 编写一个程序，要求输入一个整型命令行参数 n ，然后从标准输入读取 n 个浮点数，并打印它们的均值（平均值）和样本标准差（它们与均值的差值的平方和的平方根，除以 $n-1$ ）。
- 1.5.4 扩展前一个练习的程序，创建一个从标准输入读取 n 个浮点数的过滤器，打印出比均值大 1.5 个标准差的浮点数。
- 1.5.5 编写一个读取整数序列的程序，并打印出连续出现次数最多的整数和连续出现的次数。例如，如果输入为“1221511777711”，则程序应打印“最大连续序列是 4 个 7”。
- 1.5.6 编写一个读取整数序列并打印整数的过滤器，删除连续出现的重复值。例如，如果输入是“12215117777111111111”，你的程序应打印“1215171”。
- 1.5.7 编写一个程序，需要输入一个整型命令行参数 n ，然后读取 1 到 n 之间的 $n-1$ 个不同的整数，并确定缺少的值。
- 1.5.8 编写一个程序，从标准输入读取正浮点数，并打印其几何平均数和调和平均数。 n 个正数 x_1, x_2, \dots, x_n 的几何平均数为 $(x_1 \times x_2 \times \dots \times x_n)^{1/n}$ 。调和平均数为 $n / (1/x_1 + 1/x_2 + \dots + 1/x_n)$ 。提示：对于几何平均数，考虑采取对数以避免溢出。
- 1.5.9 假设文件 `input.txt` 只包含两个字符串“F”和“f”，下面命令的运行效果如何（参见练习 1.2.35）？

162

```
% java Dragon < input.txt | java Dragon | java Dragon
```

```
public class Dragon
{
    public static void main(String[] args)
    {
        String dragon = StdIn.readString();
        String nogard = StdIn.readString();
        StdOut.print(dragon + "L" + nogard);
        StdOut.print(" ");
        StdOut.print(dragon + "R" + nogard);
        StdOut.println();
    }
}
```

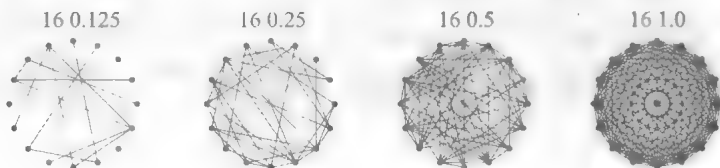
- 1.5.10 编写一个过滤器 `TenPerLine`，从标准输入读取 0 到 99 之间的整数序列并打印出来，每行 10 个整数，列对齐。然后编写一个程序 `RandomIntSeq`，它使用两个整型命令行参数 m 和 n ，并在 0 和 $m-1$ 之间打印 n 个随机整数。使用命令 `java RandomIntSeq 200 100 | java TenPerLine` 测试你的程序。
- 1.5.11 编写从标准输入读取文本的程序，并打印文本中的单词数。为了简化题目，我们约定一个单词是指由空格包围着的非空白字符序列。
- 1.5.12 编写一个程序，从标准输入读取行，每行包含一个名称和两个整数，然后使用 `printf()` 打印一个表格，一列为名称，一列为这两个整数，一列是第一个数除以第二个数的结果，精确到小数点后三位。你可以使用这样的程序制作棒球运动员的击球平均值表或学生成绩表。
- 1.5.13 编写一个程序，打印每月付款、剩余本金和贷款利息的表格，将以下三个数字作为命令行参数：年数、本金和利率（见练习 1.2.24）。
- 1.5.14 以下哪一项要求将来自标准输入的所有值保存下来（例如，使用数组来保存），哪些可以仅使用固定数量的变量来实现？对于每个选项，输入来源是指来自标准输入的 n 个实数，分布在 0 和 1 之间。
- 打印最大的数字和最小的数字。
 - 打印 n 个数字的平方和。
 - 打印 n 个数字的平均值。
 - 打印 n 个数字的中位数。
 - 打印大于平均值的数字的百分比。
 - 打印 n 个数字，以升序排列。
 - 以随机顺序打印 n 个数字。
- 1.5.15 编写一个程序，它需要三个 `double` 类型命令行参数 x 、 y 和 z ，从标准输入读取一系列点坐标的信息 (x_i, y_i, z_i) ，并打印最接近 (x, y, z) 的点的坐标。回想一下， (x, y, z) 和 (x_i, y_i, z_i) 之间距离的平方是 $(x-x_i)^2 + (y-y_i)^2 + (z-z_i)^2$ 。为了提高效率，不要使用 `Math.sqrt()`。
- 1.5.16 给定一系列物体的位置和质量，编写一个程序来计算它们中心的质量或质心。质心是 n 个物体的平均位置，以质量为单位。如果位置和质量由 (x_i, y_i, m_i) 给出，则质心 (x, y, m) 由下式得出：

$$m = m_1 + m_2 + \cdots + m_n$$

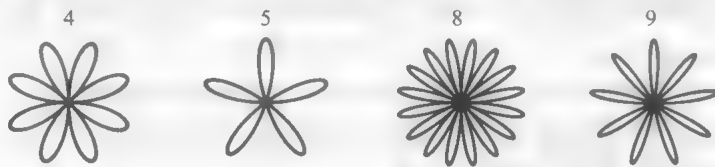
$$x = (m_1 x_1 + \cdots + m_n x_n) / m$$

$$y = (m_1 y_1 + \cdots + m_n y_n) / m$$

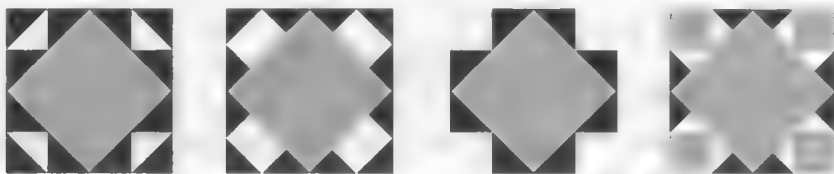
- 1.5.17 编写一个程序，读取 -1 和 +1 之间的实数序列，并打印它们的平均幅度、平均功率和零交叉数。平均幅度（average magnitude）是数据绝对值的平均值。平均功率（average power）是数据平方的平均值。零交叉数（zero crossing）是数据从严格负数转换为严格正数的次数，反之亦然。这三个统计数据被广泛用于分析数字信号。
- 1.5.18 编写一个程序，需要输入整型命令行参数 n ，并绘制一个带有红色和黑色正方形的 $n \times n$ 棋盘。将左下方的正方形设置成红色。
- 1.5.19 编写一个程序，命令行参数为整数 n 和浮点数 p （0 到 1 之间），在一个圆的圆周上绘制 n 个等间隔点，对于任意一对点，其间存在连接线的概率为 p ，则画一条灰线连接它们。



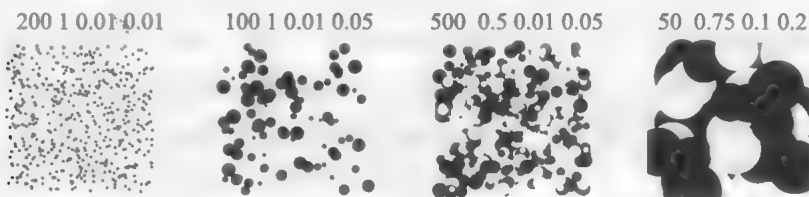
- 1.5.20 编写代码来画红桃、黑桃、梅花和方块。红桃的画法为：先画一个实心菱形，然后在左上角和右上角附上两个实心的半圆。
- 1.5.21 编写一个程序，需要一个整型命令行参数 n ，通过绘制函数 $r=\sin(n\theta)$ 从 0 到 2π 弧度的极坐标 (r, θ) 来画出 n 个花瓣（如果 n 为奇数）或 $2n$ 个花瓣（如果 n 为偶数）的玫瑰。



- 1.5.22 编写一个程序，它接受一个字符串命令行参数 s ，并将这个字符串以横幅样式显示在屏幕上，字符串从左到右移动，并在到达屏幕末尾时回到字符串开始出现的地方。添加第二个命令行参数来控制速度。
- 1.5.23 修改 PlayThatTune，添加额外的命令行参数来控制音量（将每个样本值乘以音量）和音乐速度（将每个音符的持续时间乘以速度）。
- 1.5.24 编写一个将 a.wav 文件名称和播放速率 r 作为命令行参数的程序，并以给定速率播放文件。首先，使用 StdAudio.read() 将文件读入数组 $a[]$ 。如果 $r=1$ ，则播放 $a[]$ ；否则，创建一个新的数组 $b[]$ ，其大小为 r 乘以 $a[]$ 的长度。如果 $r<1$ ，则从原始文件采样填充 $b[]$ ；如果 $r>1$ ，则从原始文件插值来填充 $b[]$ 。然后播放 $b[]$ 。
- 1.5.25 编写程序使用 StdDraw 生成以下每个图案。



- 1.5.26 编写一个名为 Circles 的程序，这个程序在单位正方形内的随机位置绘制随机半径的实心圆，产生如下所示的图像。你的程序应该有四个命令行参数：圆的数量、实心圆所占比例、最小半径和最大半径。



创新练习

- 1.5.27 可视化音频 (Visualizing audio)。修改 PlayThatTune 将要播放的值发送到标准绘图，以便你可以在播放音频时观看声波。你将必须在绘图画布上绘制多条曲线来同步声音和图像。
- 1.5.28 统计投票 (Statistical polling)。在收集某些政治民意调查的统计数据时，获得正式选民的客观样本是非常重要的。假设你有一个具有 n 个注册选民的文件，每行有一个注册选民。写一个过滤器，打印一个大小为 m 的均匀随机样本（参见程序 1.4.1）。
- 1.5.29 地形分析 (Terrain analysis)。假设地形由二维网格的高程值表示（以米为单位）。如果一个网格

点是峰值 (peak), 那么该点的四个相邻单元格 (左, 右, 上, 下) 的高程值严格低于峰值点的高程值。编写 Peaks 程序, 使其能够从标准输入中读取地形, 然后计算并打印地形中的峰值数量。

1.5.30 直方图 (Histogram)。假设标准输入流是一个 double 类型的值序列。编写一个程序, 需要整数 n 、两个实数 lo 和 hi 作为命令行参数, 将区间 (lo, hi) 分成 n 个相等区间, 统计标准输入流中的数字落在其中每个区间的数目, 使用 StdDraw 绘制一个直方图表示这个统计结果。

1.5.31 螺旋体 (Spirographs)。编写一个程序, 该程序需要三个 double 型命令行参数 R 、 r 和 a , 并绘制一个螺旋体。螺旋体图案 (技术上是外摆线) 是通过在较大的圆 (半径为 R) 的内侧滚动较小的圆 (半径为 r) 而形成的曲线。如果笔与滚动圆的中心的距离是 $(r+a)$, 那么在 t 时刻所得曲线等式由下式给出:

$$x(t) = (R+r)\cos(t) - (r+a)\cos((R+r)t/r)$$

$$y(t) = (R+r)\sin(t) - (r+a)\sin((R+r)t/r)$$

这种曲线被一种畅销的玩具所推广, 这种玩具包含边缘有齿轮的圆盘, 还有一些小孔, 你可以把一支笔放在孔里画出螺旋体的轨迹。

1.5.32 时钟 (Clock)。编写一个显示模拟时钟的秒、分、时动画的程序, 使用方法 StdDraw.pause(1000) 大约每秒更新一次显示。

1.5.33 示波器 (Oscilloscope) 编写一个模拟示波器输出的程序并产生李萨如图形 (Lissajous patterns)。这些图形是以法国物理学家 Jules A. Lissajous 的名字命名的, 他研究了当两个相互垂直的周期性振动同时发生时所产生的图案。假设输入是正弦曲线, 因此下面的参数方程描绘这个曲线:

$$x(t) = A_x \sin(w_x t + \theta_x)$$

$$y(t) = A_y \sin(w_y t + \theta_y)$$

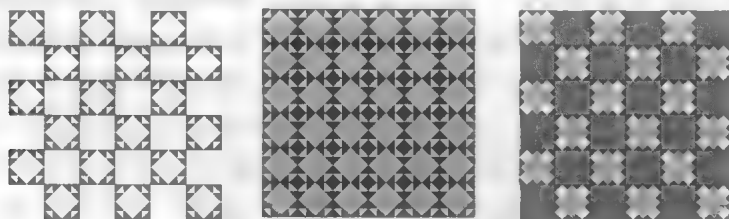
从命令行获取六个参数 A_x 、 w_x 、 θ_x 、 A_y 、 w_y 、 θ_y 。

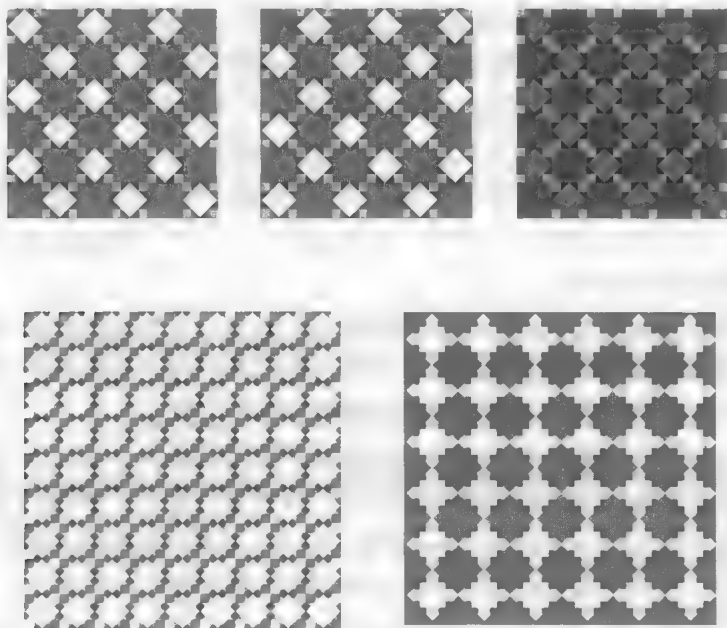
1.5.34 有轨迹的弹跳球 (Bouncing ball with tracks)。修改程序 BouncingBall, 以使其生成本书中展示的图像, 并在灰色背景上显示球的轨迹。

1.5.35 有重力的弹跳球 (Bouncing ball with gravity)。修改程序 BouncingBall, 以使其在垂直方向包含重力因素。调用 StdAudio.play(), 在球撞击墙壁时添加声音效果, 并在撞击地板时产生不同的音效。

1.5.36 随机奏乐 (Random tunes)。编写一个程序使用 StdAudio 随机产生音律。尝试通过保持一定的曲调、调高整拍的概率、增加重复以及其他规则等, 使程序尽量生成合理的旋律。

1.5.37 瓷砖图案 (Tile patterns)。使用你在练习 1.5.25 中得出的解决方案, 编写一个程序 TilePattern, 它需要一个整型命令行参数 n , 然后使用你选择的瓷砖来绘制一个 $n \times n$ 图案。第二个命令行参数用来选择棋盘格式, 第三个命令行参数用来选择颜色。使用下面的图案作为起点, 设计一个瓷砖地板。要有创意! 注意: 这些图案都是古老的设计, 你可以在许多古代 (和现代) 的建筑中找到它们。





169

1.6 案例研究：随机网络冲浪

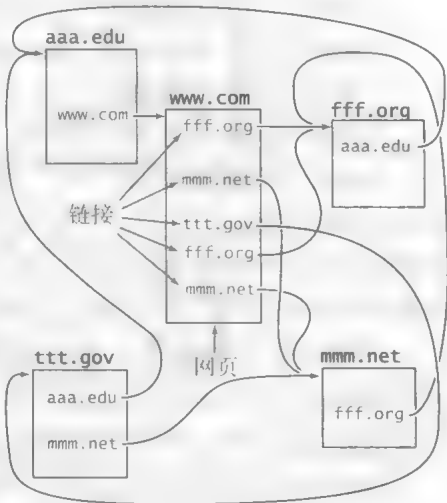
通过网络进行交流已经成为日常生活中不可或缺的一部分。我们能通过这种方式顺畅交流，在一定程度上得益于对网络结构的科学研究，这是自网络技术产生以来就在积极研究的课题。接下来我们研究一个简单的网络模型，这是一个有效地理解网络特性的方法。这种模型的变体被广泛使用，并成为网络搜索爆炸式增长的关键因素。

该模型被称为随机冲浪（random surfer）模型。这种模型非常简单。我们将网络看作是一组固定的网页，每个网页都包含一组超链接（hyperlink），每个超链接都链接到其他网页（为了简洁起见，我们使用术语网页和链接）。本节中我们研究网络冲浪者在网页间随机跳转的过程，他们通过在地址栏中输入网页名称，或者单击当前网页上的链接来跳转。

网络链接结构的基础数学模型被称为图（graph），我们将在本书后半部分（4.5 节）详细研究图及图的处理。现在我们专注于一个自然的、精心研究的概率模型相关的计算，这个模型准确地描述了随机冲浪者的行为。

研究随机冲浪模型的第一步是用公式更精确地表示它。问题的关键在于明确网页之间随机跳转的意义。接下来我们使用一种直观的 90-10 规则描述跳转到新网页的两种方法：假设随机冲浪者有 90% 的概率会通过随机点击当前网页上的链接实现跳转（每个链接以相等的概率被选中），另外 10% 的概率随机冲浪者会直接进入一个随机网页（例如，以手动输入网址的形式打开网页——译者注）（网络上所有网页被选中的概率相同）。

你很快会发现这个模型有缺陷，因为你从自身的经验中知道一个真正的网络冲浪者的行为并不是那么简单：



网页和链接

170

- 没有人以相等的概率选择链接或网页。
- 不可能以均等的概率直接浏览整个互联网上的每一个网页。
- 90-10（或任何固定）的分类只是一个猜测。
- 它不考虑后退按钮或书签。

尽管存在这些缺陷，但是这个模型足够丰富，使得计算机科学家忽略了这些缺陷而对网络的性质进行了许多研究。根据这个模型，你可以研究一下本节前文中的小例子，你认为随机冲浪者最可能访问哪个网页？

每个使用网络的人的行为都有点像随机冲浪者，所以建立网络基础设施和网络应用程序的人对了解随机冲浪者的行为非常感兴趣。该模型是了解数十亿网络用户体验的工具。在本节中，你将使用本章中学习的基本编程工具来研究该模型及其含义。

输入格式 我们希望能够在各种图上研究随机冲浪者的行为，而不仅仅是一个例子。因此，我们要编写数据驱动代码（data-driven code），保存数据文件，并编写从标准输入中读取数据的程序。这种方法的第一步是定义一个输入格式（input format），我们可以用它来构造输入文件中的信息。我们可以自由定义任何方便的输入格式。

在本书后面，将学习如何在 Java 程序中读取网页（3.1 节），并将名称转换为数字（4.4 节）或其他有效的图处理技术。现在，我们假设有 n 个网页，编号从 0 到 $n-1$ ，并且我们用有序的数字对来表示链接，如一个有序数字对 (a, b) ，是指 a 中包含指向 b 的链接。根据这些约定，随机冲浪问题的一个简单的输入格式是由一个整数（ n 的值）后跟一串整数对（表示所有的链接）组成的输入流。StdIn 将所有的空白字符序列视为一个单独的分隔符，因此

171

我们可以自由地在每行放一个或多个链接。

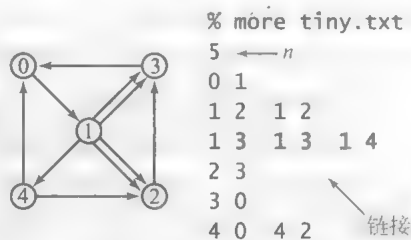
转移矩阵 我们使用一个二维矩阵来完整地指定随机冲浪者的行为，并称之为转移矩阵（transition matrix）。在 n 个网页中，我们定义一个 $n \times n$ 的矩阵，这样第 i 行第 j 列的值就是随机冲浪者从网页 i 跳转到网页 j 的概率。我们的第一个任务是编写一段代码，用于为任何给定的输入创建这样一个矩阵。按照 90-10 的规则，这个计算并不困难。我们按照如下三个步骤来计算：

- 读取 n ，然后创建数组 `counts [][]` 和 `outDegrees []`。
- 读取链接并逐渐增加计数，`counts[i][j]` 统计从 i 到 j 的链接数，`outDegrees[i]` 统计从 i 链接到的网页的总数。
- 使用 90-10 规则来计算概率。

前两个是基本的步骤，第三个步骤也不难：如果存在从 i 到 j 的链接，则用 `counts[i][j]` 乘以 $0.90/\text{outDegree}[i]$ （以 0.9 的概率取随机链接），然后每个元素加 $0.10/n$ （以 0.1 的概率进入随机网页）。Transition（程序 1.6.1）实现了上述功能，它是一个从标准输入读取图并将相关转移矩阵打印到标准输出的过滤器。

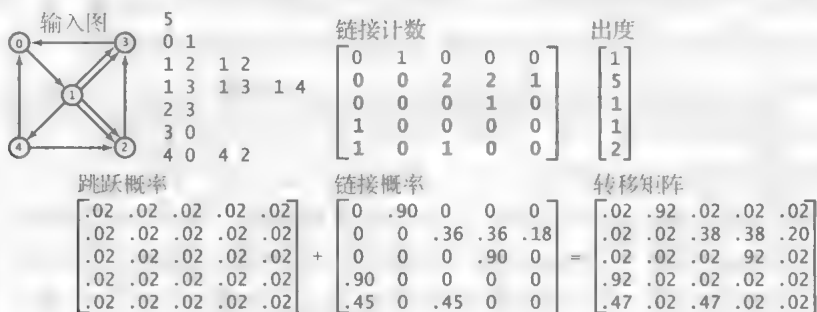
转移矩阵中所有的元素都是正数，因为每一行表示一个离散的概率分布（discrete probability distribution）——矩阵元素完全囊括了随机冲浪者下一步的所有行为，给出每个网页被访问的概率。特别要注意的是，元素总和为 1（冲浪者总是要去某个网页）。

Transition 的输出定义了另一种文件格式，可以用于描述矩阵的内容：首先输出行数 and 列数，然后按照行主映射的顺序排列逐个输出矩阵中的值（即从矩阵的左上角开始，逐行的



随机冲浪模型的输入格式

从左到右输出——译者注)。现在，我们可以编写并运行一个转移矩阵的程序。



转移矩阵的计算

172

程序1.6.1 计算转移矩阵

```

public class Transition
{
    public static void main(String[] args)
    {
        int n = StdIn.readInt();
        int[][] counts = new int[n][n];
        int[] outDegrees = new int[n];
        while (!StdIn.isEmpty())
        { // 累加链接数量
            int i = StdIn.readInt();
            int j = StdIn.readInt();
            outDegrees[i]++;
            counts[i][j]++;
        }

        StdOut.println(n + " " + n);
        for (int i = 0; i < n; i++)
        { // 打印第i行的概率分布
            for (int j = 0; j < n; j++)
            { // 打印第i行第j列的概率
                double p = 0.9*counts[i][j]/outDegrees[i] + 0.1/n;
                StdOut.printf("%8.5f", p);
            }
            StdOut.println();
        }
    }
}
  
```

n	网页数量
counts[i][j]	从网页i到网页j的链接数
outDegrees[i]	从网页i到任何网页的链接数
p	转移概率

该程序是一个从标准输入中读取链接，并在标准输出上生成相应转移矩阵的过滤器。首先它处理输入以计算每个网页的出度。然后它应用90-10规则来计算转移矩阵（参见本书中的原理描述）。它假设在输入中没有出度为0的网页（参见练习1.6.3）。

```

% more tinyG.txt
5
0 1
1 2 1 2
1 3 1 3 1 4
2 3
3 0
4 0 4 2
  
```

```

% java Transition < tinyG.txt
5 5
0.02000 0.92000 0.02000 0.02000 0.02000
0.02000 0.02000 0.38000 0.38000 0.20000
0.02000 0.02000 0.02000 0.92000 0.02000
0.92000 0.02000 0.02000 0.02000 0.02000
0.47000 0.02000 0.47000 0.02000 0.02000
  
```

173

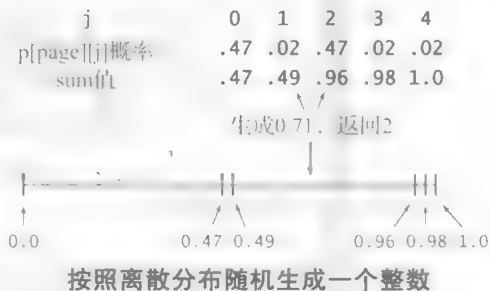
模拟 给定转移矩阵，模拟随机冲浪者行为涉及的代码非常少，正如你在 RandomSurfer（程序 1.6.2）中看到的那样。该程序根据规则从标准输入读取转移矩阵，并按照规定链接，

从第 0 页开始，以跳转次数作为命令行参数。它统计冲浪者访问每个网页的次数，除以跳转范围内的网页数量，即可得出随机冲浪者在网页上出现的概率的估计值。这个概率被称为网页的等级 (rank)。换句话说，RandomSurfer 计算所有网页等级的估计值。

一次随机跳转。计算的关键是随机跳转，这是由转移矩阵指定的。我们指定一个变量 `page`，它的值是冲浪者的当前位置。这时，转移矩阵的第 `page` 行的第 j 列表示的就是冲浪者下一步到网页 j 的概率。换句话说，当冲浪者在 `page` 上时，我们的任务是根据转移矩阵中的第 `page` 行给出的分布，生成 0 到 $n-1$ 之间的随机整数。怎样才能完成这个任务呢？我们使用一种称为轮盘赌选择 (roulettewheel selection) 的技术。我们使用 `Math.random()` 生成一个 0 到 1 之间的随机数 r ，但是它如何帮助我们进入一个随机网页呢？解决这个问题的一种方法是将第 `page` 行中的概率视为 $(0,1)$ 中的 n 个区间的集合，其中每个概率对应一个区间长度。然后我们的随机变量 r 落入其中一个区间，其概率由区间长度精确指定。根据这个推理得到下面的代码：

```
double sum = 0.0;
for (int j = 0; j < n; j++)
{ // 查找包含r的区间
    sum += p[page][j];
    if (r < sum) { page = j; break; }
}
```

变量 `sum` 追踪行 `page` 中定义的区间的端点，`for` 循环用于查找包含随机值 r 的间隔。例如，假设冲浪者在我们例子中的网页 4。转移概率为 0.47、0.02、0.47、0.02 和 0.02，`sum` 取值为 0.0、0.47、0.49、0.96、0.98 和 1.0。这些值表明概率定义了五个区间 $(0,0.47)$ 、 $(0.47,0.49)$ 、 $(0.49,0.96)$ 、 $(0.96,0.98)$ 和 $(0.98,1)$ ，每个网页一个区间。现在，假设 `Math.random()` 返回值是 0.71，我们将 j 从 0 增加到 1 再到 2 并停在那里，而 0.71 在间隔 $(0.49,0.96)$ 中，所以我们将冲浪者发送到网页 2。然后，我们开始在网页 2 执行相同的计算，随机冲浪结束并开始下一次冲浪。对于 n 较大的情况，我们可以使用二分搜索 (binary search) 来大幅加速这个计算 (见练习 4.2.38)。在这种情况下，我们愿意花更多的时间和精力来优化算法以加快搜索速度，因为我们可能需要大量的随机跳转才能完成模拟工作。



马尔可夫链。描述冲浪者行为的随机过程被称为马尔可夫链，以俄罗斯数学家安德烈·马尔可夫 (Andrey Markov) 的名字命名，他在 20 世纪初就提出了这个概念。马尔可夫链能够广泛地适用于各类问题，因此得到了深入研究，它有许多优秀以及有用的性质。例如，你可能想知道为什么 RandomSurfer 是从网页 0 开始——而非随机选择。马尔可夫链的一个基本的极限定理指出，冲浪者可以从任何地方开始，因为无论随机冲浪者从何位置开始，最终结束在任何特定页面上的概率是相同的！无论冲浪者从哪里开始，这个过程最终都会稳定到一点，并且不会为下一次冲浪提供任何信息，这种现象被称为混合 (mixing)。虽然这个现象第一感觉可能是违反直觉的，但它解释了在一个看似混乱的局面中的行为一致性。在我们这个例子中表现为，经过足够长时间的冲浪后，网络对于所有人来说看起来都是一样的。但是，并不是所有的马尔可夫链都具有这种混合属性。例如，如果我们消除了模型中的

随机跳转，网页的某些配置会给冲浪者带来问题。事实上，在网页上存在一组被称为蜘蛛陷阱（spider traps）的网页，其目的是吸引进入的链接，但没有出去的链接。没有了随机跳转，冲浪者就会被困在一个蜘蛛陷阱网页中。90-10 规则的主要目的是保证混合并消除这种异常。

程序1.6.2 模拟一个随机冲浪者

```
public class RandomSurfer
{
    public static void main(String[] args)
    { // 模拟随机冲浪
        int trials = Integer.parseInt(args[0]);
        int n = StdIn.readInt();
        StdIn.readInt();

        // 读取转移矩阵
        double[][] p = new double[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                p[i][j] = StdIn.readDouble();

        int page = 0;
        int[] freq = new int[n];
        for (int t = 0; t < trials; t++)
        { // 随机跳转到下一网页
            double r = Math.random();
            double sum = 0.0;
            for (int j = 0; j < n; j++)
            { // 查找包含r的区间
                sum += p[page][j];
                if (r < sum) { page = j; break; }
            }
            freq[page]++;
        }

        for (int i = 0; i < n; i++) // 打印网页排名
            StdOut.printf("%8.5f", (double) freq[i] / trials);
        StdOut.println();
    }
}
```

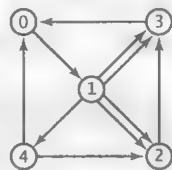
trials	跳转次数
n	网页数量
page	当前页
p[i][j]	冲浪者从第i页 跳转到第j页的 概率
freq[i]	冲浪者点击 第i页的次数

该程序使用转移矩阵来模拟随机冲浪者的行为。它将跳转次数作为一个命令行参数，读取转移矩阵，按照矩阵的规定执行指定的跳转次数，并打印每个网页的相对频率。计算的关键是随机跳转到下一页（见原文）。

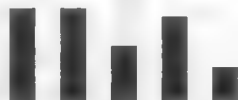
```
% java Transition < tinyG.txt | java RandomSurfer 100
0.24000 0.23000 0.16000 0.25000 0.12000
% java Transition < tinyG.txt | java RandomSurfer 1000000
0.27324 0.26568 0.14581 0.24737 0.06790
```

网页排名。RandomSurfer 模拟非常简单：它按指定的跳转次数循环，通过图随机地冲浪。由于混合现象，增加迭代次数可以更准确地估计冲浪者访问每个页面的概率（页面排名）。当你初次思考这个问题时，结果与你的直觉相比如何？你可能已经猜到，网页 4 是排名最低的网页，但是你认为网页 0 和网页 1 的排名会高于网页 3 吗？如果我们想知道哪个网页的排名是最高的，我们需要更高的精度和更高的准确性。RandomSurfer 需要 10^n 次跳转才能得到精确到 n 个小数位的答案，并且为了使这些答案稳定到一个精确值，还需要更多的跳转。就我们的例子而言，需要数万次迭代才能得到精确到小数点后两位的答案，并且要有数百万次的迭代才能得到精确到小数点后三位的答案（见练习 1.6.5）。最终的结果是网页 0 以 27.3% 的概率击败了 26.6% 概率的网页 1。如此小的差异会出现在这个问题上是相当令人惊讶的：如果你猜到了网页 0 是冲浪者最有可能去的地方，你可真幸运！

准确的网页排名估计实际上是有价值的，原因有很多。首先，与以前的方法相比，使用它们来排列符合网络搜索的结果得到的网页会更符合人们的期望。其次，这种置信度和可靠性的衡量标准为基于页面排名的网络广告带来了大量的投资。即使在我们这个微小的例子中，页面排名也可以用来说服广告客户为网页 0 支付比网页 4 多四倍的广告费。计算页面排名在数学上是合理的，它是一个有趣的计算机科学问题、一个大型的商业问题，在这里全部融合成一个问题。



0	0.273
1	0.266
3	0.146
2	0.247
4	0.068



用直方图表示的网页排名

利用直方图实现可视化。使用 StdDraw 创建可视化的表意模型也很容易，它可以让你感觉到随机冲浪的访问频率如何聚合成页面排名。如果你启用双缓冲；适当地缩放 x 坐标和 y 坐标；添加以下代码：

```
StdDraw.clear();
for (int i = 0; i < n; i++)
    StdDraw.filledRectangle(i, freq[i]/2.0, 0.25, freq[i]/2.0);
StdDraw.show();
StdDraw.pause(10);
```

到随机跳转循环；并运行 RandomSurfer 进行大量的试验，你会看到一个最终稳定的页面排名的频率直方图。一旦使用过这个工具，每当要学习一个新的模型（可能需要一些小的调整来处理更大的模型），你会发现都要用到它。

177

学习其他模型。RandomSurfer 和 Transition 是数据驱动程序中很好的例子。你可以通过创建一个像 tiny.txt 这样的文件来定义图，该文件以整数 n 开始，然后指定 0 到 $n-1$ 之间的整数对来表示页面之间的链接。鼓励按照练习中的建议运行各种数据模型，或者制作一些自己的链接图。如果你想知道网页排名是如何工作的，那么这些计算能够帮助你更好地理解一个网页的排名高于另一个网页的原因。哪种页面最可能排名更高？是有很多链接的网页，还是只有很少链接的网页？本节有很多练习研究随机冲浪者的行为。既然 RandomSurfer 使用标准输入，你也可以编写简单的程序来生成大规模的图，通过 Transition 和 RandomSurfer 用管道连接它们的输出，这样就可以研究大规模图上的随机冲浪。这种灵活性是使用标准输入和标准输出的重要原因。

直接模拟一个随机冲浪者的行为来理解网络结构是可行的，但是它有局限性。想想以下问题：你可以使用它来计算具有数百万（或数十亿！）网页和链接的网络图的网页排名吗？我们很快就可以给出否定的答案，因为你甚至不能为这么多的网页存储转移矩阵。数百万网页的矩阵将有数万亿的元素。你的计算机上有这么多空间吗？你可以使用 RandomSurfer 找到一个较小的图的网页排名吗？比如有几千个网页。要回答这个问题，你可以进行多次模拟，记录大量试验的结果，然后解释这些实验结果。我们确实使用这种方法解决了许多科学问题（赌徒破产问题就是一个例子，2.4 节将讨论另外一个问题），但这样做会非常耗时，因为需要进行大量的试验来获得所需的准确性。即使对于我们这个小例子，我们也看到需要数百万次迭代才能将页面排名精确到小数点后三位或四位。对于更大的图，获得准确估计所需的迭代次数将变得非常巨大。

178

混合马尔可夫链 页面排名是转移矩阵的一个属性，而不是将转移矩阵作为计算它们的特定方法，记住这一点非常重要，换句话说，RandomSurfer 只是一种计算页面排名的方法。

幸运的是，基于数学研究领域的简单计算模型提供了比模拟计算页面排名更有效的方法。该模型利用了我们在 1.4 节中研究过的二维矩阵的基本算术运算。

马尔可夫链的平方。随机冲浪者通过两次跳转从第 i 页跳转到第 j 页的概率是多少？第一步跳转到一个中间网页 k ，所以我们计算从 i 到 k ，然后从可能的 k 到 j 的概率，并把所有概率相加。对于我们的例子，通过两次跳转从 1 跳转到 2 的概率是从 $1 \rightarrow 0 \rightarrow 2$ 的概率 (0.02×0.02) 加上从 $1 \rightarrow 1 \rightarrow 2$ (0.02×0.38) 的概率，加上从 $1 \rightarrow 2 \rightarrow 2$ (0.38×0.02) 的概率，加上从 $1 \rightarrow 3 \rightarrow 2$ (0.38×0.02)，再加上从 $1 \rightarrow 4 \rightarrow 2$ 的概率 (0.20×0.47)，总和为 0.1172。每一对页面都有相同的计算过程。这个计算是我们以前见过的，在矩阵乘法的定义中：结果中行 i 列 j 中的元素是原始行 i 和列 j 的点积。换句话说，将 $p[i][j]$ 乘以自身的结果是一个矩阵，其中第 i 行第 j 列中的元素是随机冲浪者通过两次跳转从第 i 页跳转到第 j 页的概率。对于我们的例子，研究两次跳转转移矩阵的元素是非常值得你花费时间的，并且会帮助你更好地理解随机冲浪者的动作。例如，矩阵中的最大值是第 2 行第 0 列中的值，反映了从网页 2 开始的冲浪者只有一个到网页 3 的链接，在网页 3 只有一个到网页 0 的链接。因此，到目前为止，从第 2 页开始的冲浪者最有可能的结果是在两次跳转之后在网页 0 结束。所有其他两次跳转路线涉及其他选择的可能性较小。重要的是要注意这是一个精确的计算（仅受到 Java 浮点精度的限制）；相反，RandomSurfer 仅能产生一个估计值，需要更多的迭代来获得更准确的估计值。

幂乘方法。我们可以再次乘以 $p[i][j]$ 来计算三次跳转的概率，再乘以 $p[i][j]$ 计算四次跳转的概率，以此类推。但是，矩阵和矩阵相乘的代价是昂贵的，我们实际上需要的是向量 (vector) 和矩阵相乘，对于我们的例子，我们从向量

```
[1.0 0.0 0.0 0.0 0.0]
```

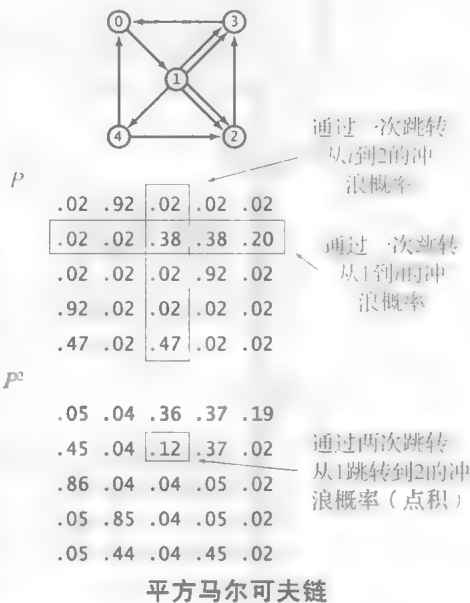
开始，它指定随机冲浪者从网页 0 开始。将这个向量与转移矩阵相乘得到向量

```
[.02 .92 .02 .02 .02]
```

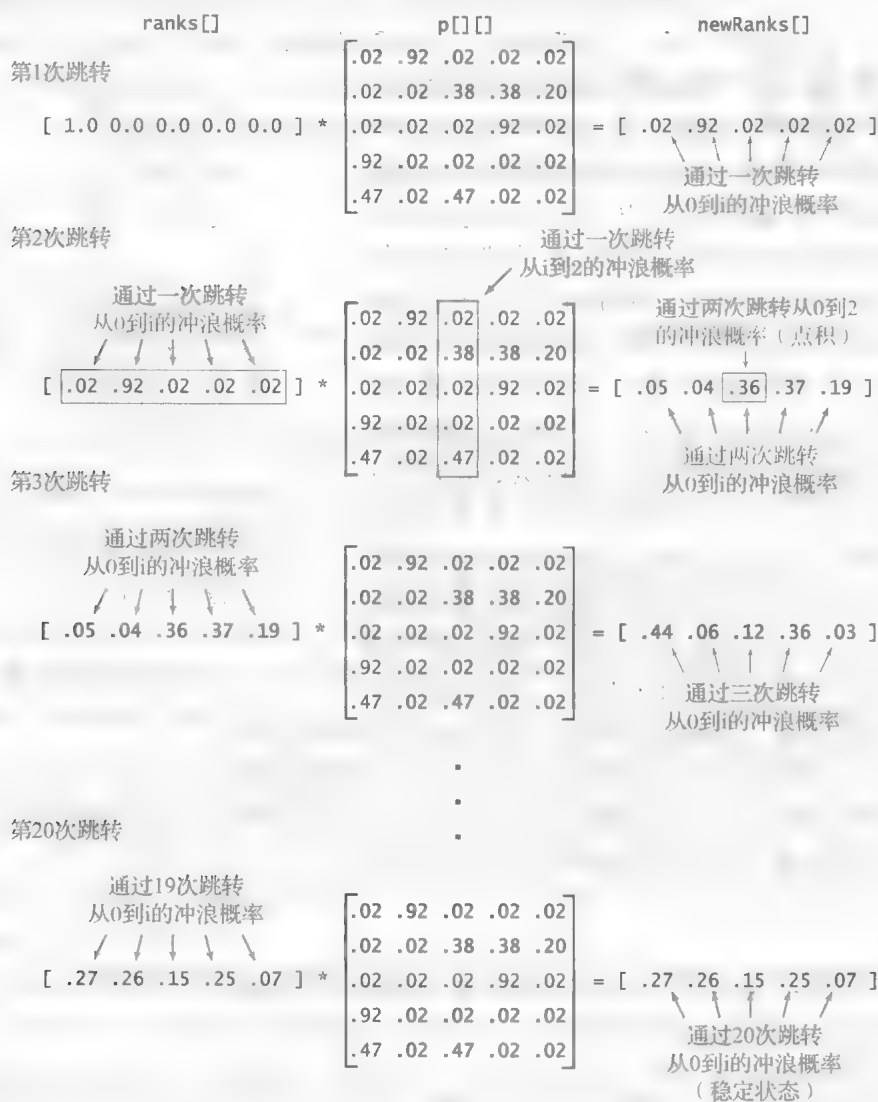
这是一次跳转后冲浪者在每个网页上结束的概率。现在，将这个向量乘以转移矩阵得到向量

```
[.05 .04 .36 .37 .19]
```

该向量包含了两次跳转后冲浪者在每个页面上结束的概率。例如，两次跳转从 0 跳转到 2 的概率是从 0 跳转到 0 到 2 (0.02×0.02) 的概率加上从 0 跳转到 1 到 2 的概率 (0.92×0.38)，加上从 0 到 2 到 2 (0.02×0.02) 的概率，加上从 0 到 3 到 2 (0.02×0.02) 的概率，再加上从 0 到 4 到 2 的概率 (0.02×0.47)，总和为 0.36。在这些初始计算中，模式是清晰的：给出随机冲浪者通过 t 次跳转之后在每个页面的概率的向量恰好是 $t-1$ 次跳转对应向量与转移矩阵的乘积。根据马尔可夫链的基本极限定理，无论我们从哪里开始，这个过程都收敛到相同的向量；换句话说，在足够次的跳转之后，冲浪者在任何给定页面上结束的概率与起点



无关。Markov（程序 1.6.3）可以用来检查我们的计算方法是不是与原来的设计一致。例如，它能够获得与 RandomSurfer 相同的结果（网页排名精确到小数点后两位），但只有 20 次矩阵 - 向量乘法运算，而不是 RandomSurfer 所需的数万次迭代。另外 20 次乘法运算给出的结果可精确到小数点后三位，而 RandomSurfer 的数百次迭代也只有少部分给出了完全精确的结果（见练习 1.6.6）。



计算网页排名的幂乘方法（转移概率的极限值）

马尔可夫链是经过精心研究的，但直到 1998 年，它对网络的影响才引起人们的关注，当时两名研究生——谢尔盖·布林（Sergey Brin）和劳伦斯·佩奇（Lawrence Page）——大胆尝试通过建立马尔可夫链来计算随机冲浪者点击整个网络（the whole web）的网页概率。它们的工作彻底改变了网络搜索的方式，这也是非常成功的网络搜索公司 Google 使用的网页排名方法的基础。具体而言，它们的想法是向用户呈现与它们的搜索查询相关的网页列表，并根据网页排名降序排列。网页排名（以及相关技术）现在占主导地位，因为在典型的搜索中，与早期的技术（如按进入链接的数量排序网页）相比，它们为用户提供了与搜索内

容更相关的网页。由于网页数量庞大，计算网页排名是一项非常耗时的工作，但它能带来巨大的效益，时间开销非常值得。

程序1.6.3 混合马尔可夫链

```
public class Markov
{ // 在测试跳转之后计算网页排名
  public static void main(String[] args)
  {
    int trials = Integer.parseInt(args[0]);
    int n = StdIn.readInt();
    StdIn.readInt();

    // 读取转移矩阵
    double[][] p = new double[n][n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        p[i][j] = StdIn.readDouble();

    // 使用幂乘方法来计算网页排名
    double[] ranks = new double[n];
    ranks[0] = 1.0;
    for (int t = 0; t < trials; t++)
    { // 计算下一次跳转对网页排名的影响
      double[] newRanks = new double[n];
      for (int j = 0; j < n; j++)
      { // 网页j的新排名是p[j][k]的
        // 第j列与之前排名的点积
        for (int k = 0; k < n; k++)
          newRanks[j] += ranks[k]*p[k][j];
      }
      for (int j = 0; j < n; j++) // 更新排名
        ranks[j] = newRanks[j];
    }
    for (int i = 0; i < n; i++) // 打印网页排名
      StdOut.printf("%8.5f", ranks[i]);
    StdOut.println();
  }
}
```

trials	跳转次数
n	网页数量
p[][]	转移矩阵
ranks[]	网页排名
newRanks[]	新的网页排名

这个程序从标准输入中读取一个转移矩阵，并计算一个随机冲浪者在命令行参数中给定的步骤数之后访问每个网页的概率（网页排名）。

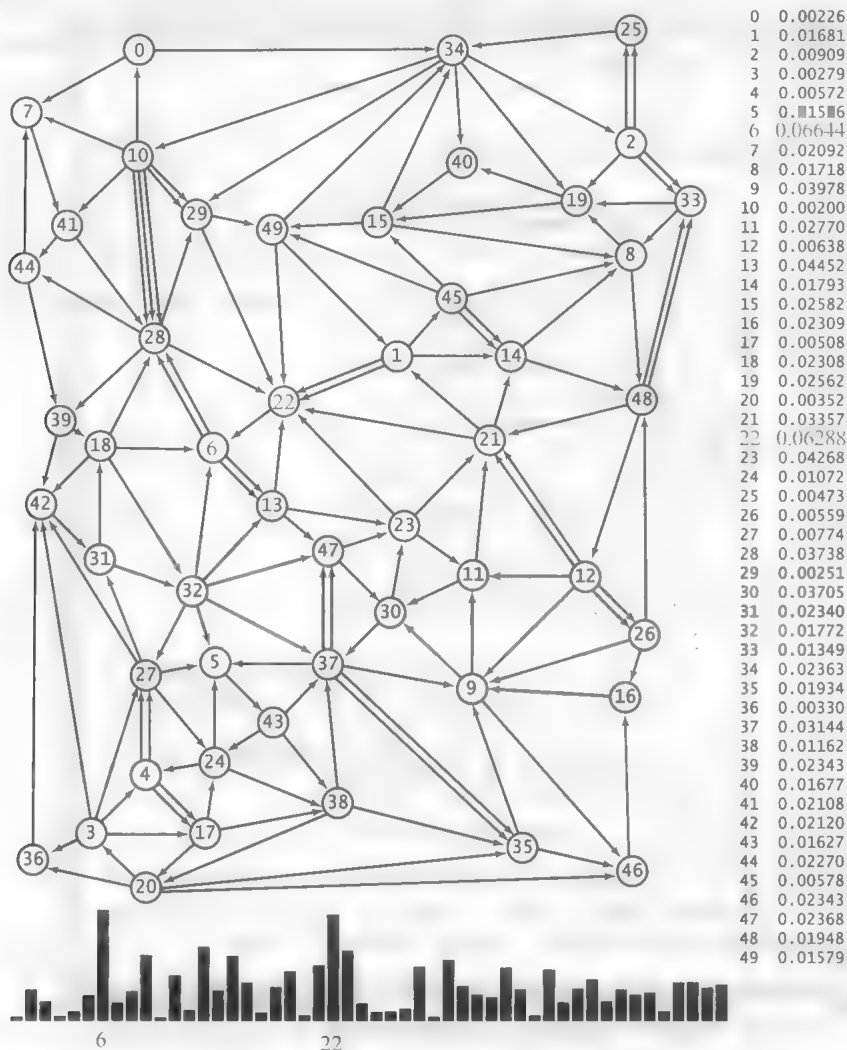
```
% java Transition < tinyG.txt | java Markov 20
0.27245 0.26515 0.14669 0.24764 0.06806
% java Transition < tinyG.txt | java Markov 40
0.27303 0.26573 0.14618 0.24723 0.06783
```

收获 对随机冲浪模型的全面理解超出了本书的范围。我们的目的是向你展示一个应用，其中包括写一个较长的程序并用这个程序来讲解特定的概念。我们可以从这个案例研究中学到哪些具体的收获呢？

我们已经有了一个完整的计算模型。数据和字符串等基本类型、条件和循环、数组和标准输入/输出/绘图/音频使你能够解决各种有趣的问题。事实上，按照理论计算机科学的基本原理，这个模型足够在任何适当的计算设备上解决任何计算问题。在接下来的两章中，我们讨论扩展模型的两个关键方法以大幅减少开发大型复杂程序所需的时间和精力。

数据驱动的代码将会大量出现。使用标准输入输出流将数据保存在文件中是一个重要的概念。我们编写了从一种输入转换为另一种输入的过滤器，能够产生用于研究的大型输入文件的生成器，以及可以处理多种模型的程序。我们可以保存数据以便当前或后续使用。我们

还可以处理来自其他来源的数据，然后将其保存在一个文件中，而无论它是来自科学仪器还是来自远程的网站。数据驱动代码概念以一种简单而灵活的方式来支持这套活动。



大规模示例的网页排名直方图

准确性不能保证。仅仅根据程序产生的结果能精确到小数点后很多位就认为结果准确是错误的。通常我们面临的最棘手的挑战是确保我们有精确的答案。

通用的随机数字生成方法只是一个开始。当我们通俗地谈论随机行为时，我们经常想到的是一些比 `Math.random()` 给我们的“每一个取值的可能性相等”更复杂的模型。我们考虑的许多问题涉及使用服从某些特定分布的随机数字（如 `RandomSurfer`）。

效率问题。盲目地认为你的计算机速度非常快，能够进行任何计算也是错误的。可能某些问题需要更多的计算工作量。例如，`Markov` 中使用的方法比直接模拟随机冲浪者的行为更有效率，但是对于实际出现的巨大的 `Web` 图来说，其计算网页排名的速度仍然太慢。第4章专门讨论如何评估你编写的程序的性能。在此之前，我们推迟对这些问题的详细研究，但请记住，你始终需要对你的程序的性能要求有一些总体的了解。

或许我们要学习的最重要的收获是：编写像本节练习那样的复杂问题的程序时，调试程

序难度很高。本书中精心设计的程序掩盖了这一收获，每一个程序都经过了长时间的测试，不断地修复漏洞，并在大量的输入中运行过。一般而言，我们在本书的讲解中尽量避免描述编程错误以及修改过程，否则会让学习过程变得很枯燥，或让你的注意力过多地集中在错误代码上。练习和本书官网中有一些错误代码示例和修正过程的描述。

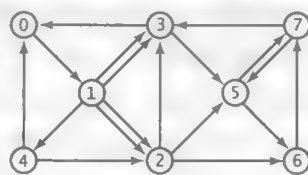
185

练习

- 1.6.1 修改 Transition 将跳转概率作为命令行参数，并使用你修改的版本计算切换到 80-20 规则或 95-5 规则的网页排名效果。
- 1.6.2 修改 Transition 以忽略多个链接的影响。也就是说，如果从一个网页到另一个网页有多个链接，请将它们计为一个链接。创建一个小例子，展示这个修改如何改变网页排名。
- 1.6.3 修改 Transition 来处理没有跳出链接的网页，把这些网页在转移矩阵中对应的一行全部设置为 $1/n$ ，其中 n 是列的数量。
- 1.6.4 RandomSurfer 中的代码段产生随机跳转时如果行 p [page] 的概率加起来不等于 1 则失败。请解释在这种情况下发生了什么，并提出解决问题的方法。
- 1.6.5 尝试估算对于 tiny.txt 文件所描述的图，RandomSurfer 计算精确到小数点后 4 位和小数点后 5 位的网页排名所需要的迭代次数。
- 1.6.6 判断用 Markov 方法计算 tiny.txt 精确到小数点后 3 位、小数点后 4 位和小数点后 10 位的网页排名需要的迭代次数。
- 1.6.7 从本书官网下载文件 medium.txt（用本节描述格式保存的 50 个网页的信息），添加从网页 23 到其他每一个网页的链接。观察它对网页排名的影响，并讨论结果。
- 1.6.8 向 medium.txt 添加（请参阅前面的练习）从其他每一个网页到网页 23 的链接，观察它对网页排名的影响，并讨论结果。
- 1.6.9 假设你的网页是在 medium.txt 中的网页 23。有没有一个从你的网页到其他网页的链接：添加这个链接后可以提高你的网页排名？
- 1.6.10 假设你的网页是在 medium.txt 中的网页 23。有没有一个从你的网页到其他网页的链接：添加这个链接后可以降低你的网页排名？

186

- 1.6.11 使用 Transition 和 RandomSurfer 来确定下面展示的具有 8 个网页的图的网页排名。
- 1.6.12 使用 Transition 和 Markov 来确定下面展示的具有 8 个网页的图的网页排名。



8 个网页的图的示例

187

创新练习

- 1.6.13 矩阵平方。编写一个像 Markov 这样的程序，通过重复对矩阵进行平方来计算网页排名，从而计算序列 p 、 p^2 、 p^4 、 p^8 、 p^{16} 等。验证矩阵中的所有行是否收敛到相同的值。
- 1.6.14 随机网络。为 Transition 创建一个生成器，将网页计数 n 和链接计数 m 作为命令行参数，并向标准输出打印 n ，后面跟随 m 个取值范围为 0 到 $n-1$ 的随机整数对。（关于更真实的 Web 模型的讨论，请参阅 4.5 节）。
- 1.6.15 中心网页与发散网页。向上一个练习中的生成器添加固定数量的中心网页，即随机选取的 10% 网页有链接指向这些中心网页，以及添加固定数量的发散网页，即它们有指向随机选取的 10% 网页的链接。计算网页排名。哪个排名更高，中心网页还是发散网页？

- 1.6.16 网页排名。设计一个图，其中指向排名最高的网页的链接数比指向其他网页的链接数更少。
- 1.6.17 点击次数。一个网页的点击次数是指随机冲浪者访问该网页的跳转次数。运行实验来估计 `tiny.txt` 的点击次数，将点击次数与网页排名进行比较，构想一个关于这种关系的假设，然后在 `medium.txt` 上验证你的假设。
- 1.6.18 覆盖时间。编写一个程序，从一个随机的网页开始，估计随机冲浪者每个网页都至少访问一次所需的时间。
- 1.6.19 图形模拟。创建一个图形模拟网页排名，其中代表每个页面的点的大小与其页面排名成比例。为了使你的程序由数据驱动，设计一个文件格式，其中包括指定每个网页应绘制的位置的坐标。在 `medium.txt` 上测试你的程序。

函数和模块

本章重点介绍一种重要的代码结构：**函数 (function)**。与条件和循环一样，函数对控制流具有深远影响，可以用于在不同代码段之间来回传递控制。函数（在 Java 中称为静态方法）的重要性不言而喻，因为它们可以在程序中明确地分割任务，而且其提供了一种机制使我们能够重复使用功能类似的代码。

我们将函数集中在可以独立编译的模块 (**module**) 中。我们使用模块将计算编程任务分解成合理大小的子任务。你将在本章中学习如何构建你自己的模块以及如何使用它们，以编程的风格划分，这称作模块化编程 (**modular programming**)。

模块被建立的主要目的是为了代码复用，就是使得编写出的代码可以在稍后的许多其他程序中再次使用。我们将这些模块称为库 (**library**)。在本章中，尤其重要的是用于生成随机数、分析数据以及为数组提供输入输出的库。库的使用极大地扩展了程序本身提供的基本数据操作。

将控制传递给函数自身的过程称为递归。起初，递归可能看起来违反直觉，但当你理解它的原理后，可以用它开发简单的程序以实现复杂的任务，而这些复杂的任务本来可能是非常难于实现的。

在编程时，要尽可能地将程序明确地分割成相互独立的子任务，然后分别实现。本章中我们会不断地强调这一点，并在本章最后一个案例中展示如何通过将复杂的编程任务分解成更小的子任务，然后独立开发以实现子任务，并编写彼此交互的模块，最终完成复杂的编程任务。

190
191

2.1 函数的定义

用于实现函数的 Java 结构称为静态方法 (**static method**)。形容词 **static** 是为了将这种方法与第 3 章中将要讨论的方法区分开——我们从现在开始将一直使用“静态”这个形容词，但是到第 3 章才讨论这两类方法的差异。其实从本书一开始，你就已经使用了静态方法，从数学函数库中的 `Math.abs()` 和 `Math.sqrt()`，到 `StdIn`、`StdOut`、`StdDraw` 和 `StdAudio` 中的所有方法。事实上，你编写的每个 Java 程序都有一个名为 `main()` 的静态方法。在本节中，你将学习如何定义自己的静态方法。

在数学中，函数的作用是将输入值（一种类型或取值范围）映射到输出值（另一种类型或取值范围）。例如，函数 $f(x)=x^2$ 映射 2 到 4、3 到 9、4 到 16 等。首先，我们使用静态方法来实现一些数学函数，因为这些函数我们比较熟悉。Java 的数学库中实现了许多标准的数学函数，但是科学家和工程师们会使用各种各样的数学函数，这些函数不会全部包含在库中。在本节的开头，你将学习如何自行实现这些函数。

之后，你将了解到，静态方法可以实现的远不止数学函数的功能，例如，静态方法的取值范围可以是字符串或其他类型，也可以实现如打印输出等副功能。我们还在本节中考虑如何使用静态方法组织程序，从而简化复杂的编程任务。

静态方法支持一个关键概念：在编程时，要尽可能地将程序明确地分割成相互独立的子任务，然后分别实现。这是一种重要的编程方法，从现在开始你就应该记住并执行。我们将

会在本节中不断地强调、强化这一点。当你写一篇文章时，你将其分解成段落；当你编写程序时，你会把它分解成方法。在编程中将较大的任务分成较小的任务甚至比编写代码本身都要重要得多，因为它大大方便了调试（debugging）、维护（maintenance）和复用（reuse），这些都是开发良好软件的关键。

192

静态方法 正如你从使用 Java 数学库所知道的，静态方法的使用很容易理解。例如，当你在程序中编写 `Math.abs(a-b)` 时，效果就如同你将代码替换为 Java 的 `Math.abs()` 方法的返回值，其中 `a-b` 是方法的参数。这种用法是非常直观的，我们几乎不需要讨论它。如果你想知道系统怎样实现这个效果，你会看到它实际上改变了程序的控制流（control flow）。通过这种方式改变控制流，与通过条件和循环改变控制流是同等重要的。

在 .java 文件中，你可以随意为你的静态方法命名（除了 `main()` 方法以外），然后就可以用一系列语句代码实现方法的具体功能。我们稍后会研究细节，在这里我们先使用一个简单的例子——**Harmonic**（程序 2.1.1）——说明函数如何影响控制流。在这个程序中有一个名为 `harmonic()` 的静态方法，需要输入一个整数参数 `n`，并返回第 `n` 个谐波数（参见程序 1.3.5）。

程序 2.1.1 优于我们原来的计算谐波数的程序（程序 1.3.5），因为它清楚地将程序分为了两个主要任务：计算谐波数并与用户进行交互（为了说明，程序 2.1.1 需要几个命令行参数，而不是只有一个）。在编程时，要尽可能地将程序明确地分割成相互独立的子任务，然后分别实现。

```
public class Harmonic
{
    public static double harmonic(int n)
    {
        double sum = 0.0;
        for (int i = 1; i <= n; i++)
            sum += 1.0/i;
        return sum;
    }

    public static void main(String[] args)
    {
        for (int i = 0; i < args.length; i++)
        {
            int arg = Integer.parseInt(args[i]);
            double value = harmonic(arg);
            StdOut.println(value);
        }
    }
}
```

调用静态方法的控制流

程序2.1.1 谐波数（二）

```
public class Harmonic
{
    public static double harmonic(int n)
    {
        double sum = 0.0;
        for (int i = 1; i <= n; i++)
            sum += 1.0/i;
        return sum;
    }

    public static void main(String[] args)
    {
        for (int i = 0; i < args.length; i++)
        {
            int arg = Integer.parseInt(args[i]);
            double value = harmonic(arg);
            StdOut.println(value);
        }
    }
}
```

sum | 累计总和

arg | 参数
value | 返回值

该程序定义了两个静态方法，一个名为 `harmonic()`，它需要输入整数参数 `n`，并计算第 `n` 个谐波数（见程序 1.3.5），另一个命名为 `main()`，它用命令行指定的整数参数来调用 `harmonic()`。

```
% java Harmonic 1 2 4
1.0
1.5
2.0833333333333333
```

```
% java Harmonic 10 100 1000 10000
2.9289682539682538
5.187377517639621
7.485470860550343
9.787606036044348
```

控制流。既然 Harmonic 程序实现的是一个我们熟悉的数学函数，那么我们就可以仔细地研究它的代码，以便仔细思考什么是静态方法，以及静态方法的运行方式。Harmonic 包含两个静态方法：harmonic() 和 main()。即使 harmonic() 在代码中首先出现，但 Java 执行的第一个语句仍然是 main() 中的第一个语句。接下来的几个语句照常运行，除了每次遇到调用静态方法 harmonic() 的语句时，即代码 harmonic(arg)，会导致将控制流转移到 harmonic() 中的第一行代码。而且，在调用时，Java 将 harmonic() 中的参数变量 n 初始化为 main() 中的 arg 值。那么，Java 就可以照常执行 harmonic() 中的语句，直到它到达一个 return 语句，其才将控制传递给 main() 中包含对 harmonic() 的调用的语句。此外，调用 harmonic(arg) 方法将会产生一个值——该值由 return 语句指定，在 harmonic() 的代码中执行 return 语句返回的变量是 sum。然后 Java 将此返回值赋给变量 value。最终结果与我们的直觉完全吻合：第一次赋值给 value 的值是 1.0，第一次打印出来的值也是 1.0——这个值就是参数变量 n 被初始化为 1 时，由 harmonic() 计算的值。第二次为 value 赋值并打印出的值为 1.5——即当 n 初始化为 2 时，由 harmonic() 计算的值。每个命令行参数重复相同的过程，在 harmonic() 和 main() 之间来回传递控制。

跟踪函数调用的过程。通过跟踪来分析函数调用控制流的一个简单方法是，假设每个函数在被调用时都输出其名称和参数值，在返回之前显示它的返回值，并在被调用时添加缩进，返回时取消缩进。这是我们自 1.2 节以来一直使用的方法，通过打印其变量的值来对程序实现跟踪，而且现在，我们要加强这个跟踪方法，把函数调用的过程也显示出来。添加的缩进将显示控制流，并帮助我们检查每个函数是否具有我们预期的效果。通常，用这种方式添加一些 StdOut.println() 语句来跟踪程序的执行，这可以用于分析任何程序的控制流，是用于分析程序功能和原理的好方法。如果返回值符合我们的期望，我们不需要详细地跟踪这个函数代码，从而为我们节省了大量的工作。

对于本章的其余部分，你的编程工作将主要围绕创建和使用静态方法展开，因此有必要更详细地研究其基本属性。随后，我们将研究几个关于函数实现和应用的例子。

术语。将抽象概念和 Java 的具体实现机制进行区分对于理解它们是很有帮助的（例如，Java 的 if 语句实现了条件分支，while 语句实现了循环，以此类推）。以下表格总结了一些数学函数概念和 Java 结构。表格中的这些函数已经为数学家使用了几百年、程序员使用了几十年，因此我们不会详细地研究函数实现的具体细节，而只是关注那些有利于我们编程的部分。

当我们在定义数学函数的公式（如 $f(x)=1+x+x^2$ ）时，公式中使用的符号名称 x 是输入值的占位符，在计算时它将被输入变量替换，并通过公式计算确定输出值。在 Java 中，我们使用一个参数变量（parameter variable）作为符号占位符，当函数被调用时，将特定输入值作为参数（argument）为其赋值并进行求值计算。

```
i = 1
arg = 1
harmonic(1)
    sum = 0.0
    sum = 1.0
    return 1.0
value = 1.0
i = 2
arg = 2
harmonic(2)
    sum = 0.0
    sum = 1.0
    sum = 1.5
    return 1.5
value = 1.5
i = 3
arg = 4
harmonic(4)
    sum = 0.0
    sum = 1.0
    sum = 1.5
    sum = 1.8333333333333333
    sum = 2.0833333333333333
    return 2.0833333333333333
value = 2.0833333333333333
```

Java Harmonic 参数为

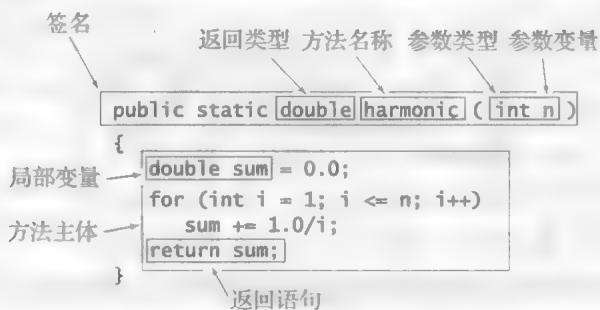
1、2、4 时的函数调用跟踪

193
194

195

概念	Java 结构	描述
函数	静态方法	映射
输入值	参数	输入到函数
输出值	返回值	函数输出
公式	方法主体	函数定义
自变量	参数变量	输入值的符号占位符

静态方法定义。静态方法定义的第一行（称为签名），用于为方法和每个参数变量提供一个名称。它还指定每个参数变量的类型和方法的返回类型。签名由关键字 **public**、关键字 **static**、返回类型、方法名称组成，也可以包括一系列参数变量。参数变量可以有零个或多个，用逗号分隔，并括在括号中，标明变量类型和名称。我们将在下一节中讨论 **public** 关键字的含义，在第3章中讨论 **static** 关键字的含义（从技术上讲，Java 中的签名可以仅包括方法名称和参数类型，但是我们认为此种表示方式仅限专业人员使用）。签名后是方法主体，包含在大括号中。主体由我们在第1章中讨论的各种语句组成。它可以包含一个返回语句，它将控制传递回静态方法被调用的位置，并返回计算结果或返回值。主体也可以声明局部变量，这些变量只能在声明方法的内部使用。



静态方法剖析

函数调用。正如你已经看到的，Java 中的静态方法调用只不过是方法名称；其紧跟着用逗号分隔并括在括号中的参数，这与数学函数通常的格式完全相同。如 1.2 节所述，方法调用是一个表达式，因此你可以使用它来构建更复杂的表达式。同样，参数是一个表达式——Java 可以计算表达式，并将结果传递给方法。所以，你可以编写像 `Math.exp(-x*x/2) / Math.sqrt(2*Math.PI)` 这样的代码，Java 能够知道你的意思。

```
for (int i = 0; i < args.length; i++)
{
    arg = Integer.parseInt(args[i]);
    double value = harmonic(arg);
    StdOut.println(value);
}
```

↑ 函数调用
参数

函数调用剖析

多个参数。与数学函数一样，Java 静态方法可以接受多个参数，因此可以有多个参数变量。例如，下面的静态方法用直角三角形的直角边 *a* 和 *b* 计算其斜边的长度：

```
public static double hypotenuse(double a, double b)
{ return Math.sqrt(a*a + b*b); }
```

虽然在这种情况下参数变量的类型是相同的，但通常它们可以是不同的类型。每个参数变量的类型和名称在函数签名中声明，每个声明用逗号分隔。

多个方法。你可以在 `.java` 文件中定义尽可能多的静态方法。每个方法都有一个由大括号括

起来的语句序列组成的主体。这些方法是独立的，可以以任何顺序出现在文件中。静态方法可以调用在同一个文件中的任何其他静态方法或 Java 库中的任何静态方法（如 Math），如下所示：

```
public static double square(double a)
{ return a*a; }

public static double hypotenuse(double a, double b)
{ return Math.sqrt(square(a) + square(b)); }
```

另外，我们在下一节中可以看到，静态方法可以调用其他 .java 文件中的方法（只要 Java 可以访问那些文件）。在 2.3 节中，我们将研究的静态方法甚至可以调用自己。

197

重载。静态方法因其签名不同而不同。例如，我们经常想为不同数值类型的值定义相同的操作，如以下用于计算绝对值的静态方法：

```
public static int abs(int x)
{
    if (x < 0) return -x;
    else      return x;
}

public static double abs(double x)
{
    if (x < 0.0) return -x;
    else        return x;
}
```

这是两个不同的方法，但是足够相似，因其使用了相同的名称（abs）。对于两个静态方法，如果签名内容不同而使用了相同的名称，则称之为重载（overloading），这在 Java 编程中较为常见。例如，Java Math 库使用此方法为所有基本数值类型提供 Math.abs()、Math.min() 和 Math.max() 的实现。重载的另一个常见用法是定义方法的两个不同版本：一个使用参数，另一个使用该参数的默认值。

多个返回语句。你可以将 return 语句放在任何位置：一旦到达第一个 return 语句，控制将返回到调用程序。下面这个素数测试函数就是一个使用多个返回语句的例子：

```
public static boolean isPrime(int n)
{
    if (n < 2) return false;
    for (int i = 2; i <= n/i; i++)
        if (n % i == 0) return false;
    return true;
}
```

即使可能有多个 return 语句，但所有静态方法都会在每次调用时返回单个值：遇到的第一个返回语句后面的值。有些程序员坚持每个方法应该只有一个返回语句，但是我们在本书中没有那么严格。

198

整型值的绝对值

```
public static int abs(int x)
{
    if (x < 0) return -x;
    else      return x;
}
```

实现函数的典型代码（静态方法）

双精度浮点 型值的绝对值	<pre> public static double abs(double x) { if (x < 0.0) return -x; else return x; } </pre>
素数检测	<pre> public static boolean isPrime(int n) { if (n < 2) return false; for (int i = 2; i <= n/1; i++) if (n % i == 0) return false; return true; } </pre>
计算直角 三角形的斜边	<pre> public static double hypotenuse(double a, double b) { return Math.sqrt(a*a + b*b); } </pre>
谐波数	<pre> public static double harmonic(int n) { double sum = 0.0; for (int i = 1; i <= n; i++) sum += 1.0 / i; return sum; } </pre>
[0, n) 中的 均匀随机整数	<pre> public static int uniform(int n) { return (int) (Math.random() * n); } </pre>
绘制一个三角形	<pre> public static void drawTriangle(double x0, double y0, double x1, double y1, double x2, double y2) { StdDraw.line(x0, y0, x1, y1); StdDraw.line(x1, y1, x2, y2); StdDraw.line(x2, y2, x0, y0); } </pre>

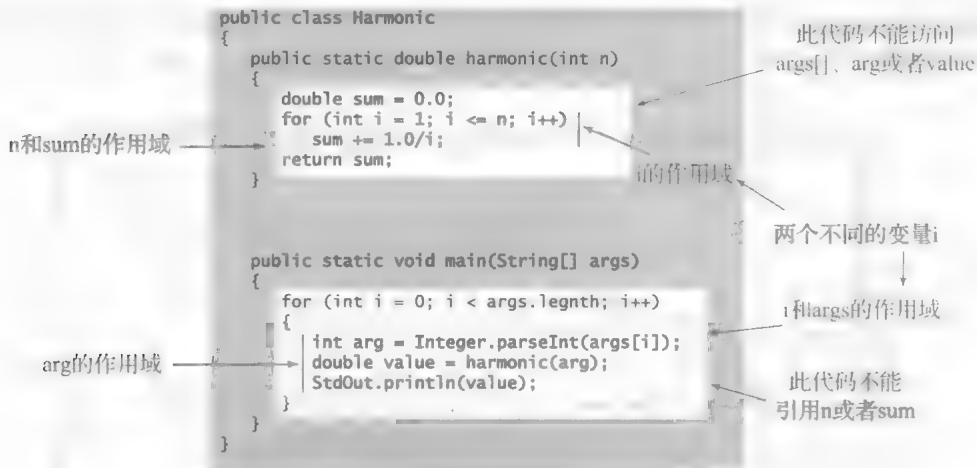
实现函数的典型代码（静态方法）

199

单个返回值。Java 方法只给调用者提供一个返回值，它的类型就是方法签名中声明的类型。这个策略并不像它看起来那么严格，因为 Java 数据类型不仅限于基本数据类型，因此可以包含更多的信息。本节后半段，你可以看到数组被用作返回值。

作用域。变量的作用域是指能够依据其变量名引用这一变量的代码范围。通常 Java 的规则是：在一个语句块中声明的变量，其作用域仅限于该块中的语句。尤其是，静态方法中声明的变量的范围仅适用于该方法的主体。因此，你不能在一个静态方法中引用另一个静态方法中声明的变量。如果方法中包含了较小的块，如 if 或 for 语句体，则在其中一个块中声明的任何变量的作用域仅限于该块的内部。在实际的编程工作中，常常会出现在独立的代码块中使用相同变量名的情况。当我们这样做时，我们要清楚这些其实是不同的变量。例如，当我们在同一程序中的两个不同的 for 循环中使用索引 i 时，我们一直遵循这种做法。设计软件时的指导原则是在声明每个变量时应该使其作用域尽可能小。我们使用静态方法的一个重要原因就是它们能够通过限制变量作用域来简化调试过程。

副作用。在数学中，函数将一个或多个输入值映射到某个输出值。在计算机编程中，许多函数也遵从这一模型：它们接受一个或多个参数，并产生单个返回值。纯函数是这样一种函数，即给定相同的参数，总是返回相同的值，而不产生任何可见的副作用，这些副作用可能是消耗输入、产生输出或以其他方式更改系统的状态。函数 harmonic()、abs()、isPrime() 和 hypotenuse() 都是纯函数。



局部变量和参数变量的作用域

200

然而，在计算机编程中，产生副作用有时也是有益的。事实上，我们经常定义只产生副作用的函数。在 Java 中，静态方法可以使用关键字 `void` 作为其返回类型，以指示它没有返回值。`void` 类型的静态方法中不需要显式调用 `return` 语句：在 Java 执行方法的最后一个语句之后，控制返回给调用者。

例如，静态方法 `StdOut.println()` 的副作用是将给定参数打印到标准输出（这个函数没有返回值，副作用也可以说是它的主作用）。类似地，以下静态方法的副作用是将三角形绘制到标准绘图（并且没有指定的返回值）：

```

public static void drawTriangle(double x0, double y0,
                                double x1, double y1,
                                double x2, double y2)
{
    StdDraw.line(x0, y0, x1, y1);
    StdDraw.line(x1, y1, x2, y2);
    StdDraw.line(x2, y2, x0, y0);
}

```

从编程技巧上来说，一般编写一个既产生返回值又能够产生副作用的静态方法并非良策。一个典型的例外是读取输入的函数。例如，`StdIn.readInt()` 返回一个值（一个整数）并产生副作用（从标准输入中消耗一个整数）。在本书中，我们使用 `void` 静态方法有两个主要目的：

- 对于 I/O，使用 `StdIn`、`StdOut`、`StdDraw` 和 `StdAudio`。
- 操纵数组的内容。

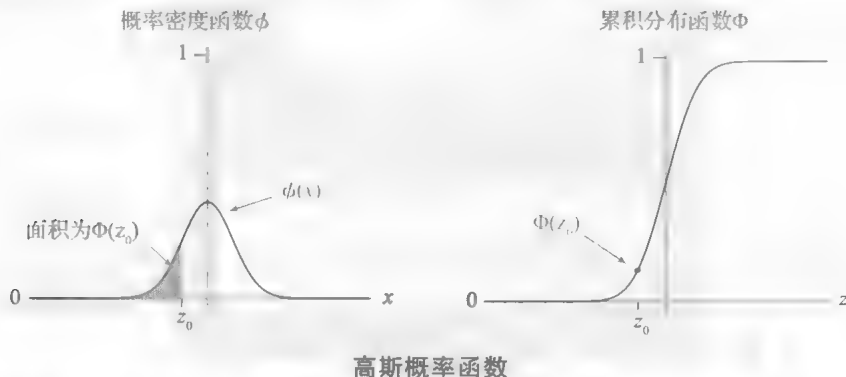
从 `HelloWorld` 中的 `main()` 开始，你一直在使用 `void` 类型的静态方法，我们将在本节稍后讨论它们在数组中的用法。从第 3 章开始，我们将以 Java 支持的特定方式，在 Java 中编写具有其他副作用的方法。

201

实现数学函数 为什么不直接使用 Java 中定义的方法，如 `Math.sqrt()`？答案是：当 Java 库里存在时，我们可以使用；但我们可能希望使用的数学函数是无限数量的，而库中只有很小的一部分。当你使用不在库中的数学函数时，需要实现相应的静态方法。

作为示例，我们考虑一段常用而且重要的代码，相信许多美国高中生和大学生会对其颇感兴趣。近一年来，有 100 多万名学生参加了“美国高考”。因为学生可以选择参加不同的考试，因此学生的分数从 400 分（最低）到 1600 分（最高）不等。这些分数在做出一些重要决定时起到关键的作用：例如，学生运动员必须获得至少 820 分，申请某些学术奖学金的最

低资格要求为 1500 分。那么，在美国的高考中，有多少人不符合学生运动员的要求？有多少人可以拿到奖学金？



高斯概率函数

统计学的两个函数使我们能够准确地回答这些问题。高斯（正态）概率密度函数（Gaussian (normal) probability density function）的函数曲线相信大家都很熟悉，其公式为 $\phi(x) = e^{-x^2/2} / \sqrt{2\pi}$ 。高斯累积分布函数（Gaussian cumulative distribution function） $\Phi(z)$ 是指由 $\phi(x)$ 定义的曲线下方 x 轴上方、垂直线 $x=z$ 的左侧区域的面积。这两个函数因其对自然世界的准确描述和对实验错误的纠正，在科学、工程和金融领域起到了重要作用。

特别需要说明的是，这两个函数可以用于准确地描述本例中分数的分布（其值每年都会公布），并且可以描述为平均值（分数的平均值）和标准差（每个分数和平均值之间差异的平方和再求平方根）的函数。知道了平均值 μ 和考试成绩的标准差 σ ，通过函数 $\Phi((z-\mu)/\sigma)$ 可计算分数小于给定值 z 的学生的百分比。计算 ϕ 和 Φ 的静态方法在 Java 的 Math 库中不可用，因此我们需要开发自己的函数。

封闭形式。如果我们用一个公式来定义我们需要的函数，最简单的情况下，公式中的每一个组成元素都可以用 Java 函数库中的函数直接实现。这种情况称为封闭形式。 ϕ 就是这种情况——Java 的 Math 库中包含了计算指数和平方根函数的方法（以及常量 π 的值），因此，与其数学定义相对应的静态方法 `pdf()` 是易于实现的（参见程序 2.1.2）。

非封闭形式。在某些情况下，我们可能需要更复杂的算法来计算函数值。对于 Φ 的实现就是这种情况——此函数没有封闭形式的表达式。为了近似估算函数的值，有时会使用泰勒级数逼近法。实际上，为数学函数建立可靠而精确的代码实现是科学和艺术的结合，有时需要使用过去几个世纪积累的数学知识。已经有多种不同的方法可以用于近似估算 Φ 。例如，可以使用一个泰勒级数来逼近 Φ 和 ϕ 的比率，从而得出以下计算关系：

$$\Phi(z) = \frac{1}{2} + \phi(z) \left(z + \frac{z^3}{3} + \frac{z^5}{(3 \cdot 5)} + \frac{z^7}{(3 \cdot 5 \cdot 7)} + \dots \right)$$

此公式可以轻易转换为程序 2.1.2 中的静态方法 `cdf()` 的 Java 代码。对于 z 较小的情况，该值非常接近于 0，因此代码直接返回 0；对于 z 较大的情况，该值非常接近于 1，因此代码直接返回 1；否则，通过不断展开泰勒级数的项来计算上述关系式，直到计算的结果收敛（即展开后新增加的泰勒级数项的值已经不再影响计算结果——译者注）。

使用适当的参数在命令行中运行 Gaussian 程序，可以得到约 17% 的考生不符合体育运动资格，只有约 1% 的有资格获得奖学金。在平均值为 1025 和标准差为 231 的一年中，约有 2% 的人有获得奖学金资格。

各种数学函数的计算一直在科学和工程中起着重要的作用。在许多应用程序中，有时候你需要的函数存在于 Java 的数学库中，如我们刚刚看到的 `pdf()`；有时候可以简单地使用泰

勒级数逼近进行计算，如我们刚刚看到的 `cdf()`。事实上，实现这类计算任务在计算系统和编程语言的演进过程中发挥了核心作用。你将在本书官网和本书中找到许多例子。

203
204

程序2.1.2 高斯函数

```
public class Gaussian
{ // 高斯（正态）概率密度函数的实现
  public static double pdf(double x)
  {
    return Math.exp(-x*x/2) / Math.sqrt(2*Math.PI);
  }

  public static double cdf(double z)
  {
    if (z < -8.0) return 0.0;
    if (z > 8.0) return 1.0;
    double sum = 0.0;
    double term = z;
    for (int i = 3; sum != sum + term; i += 2)
    {
      sum = sum + term;
      term = term * z * z / i;
    }
    return 0.5 + pdf(z) * sum;
  }

  public static void main(String[] args)
  {
    double z = Double.parseDouble(args[0]);
    double mu = Double.parseDouble(args[1]);
    double sigma = Double.parseDouble(args[2]);
    StdOut.printf("%.3f\n", cdf((z - mu) / sigma));
  }
}
```

sum	累计总和
term	当前值

该代码实现了高斯概率密度函数（pdf）和高斯累积分布函数（cdf），这些函数并未在Java的Math库中。计算pdf()直接来自其定义，计算cdf()需要使用泰勒级数，并调用pdf()（参见补充内容和练习1.3.38）。

```
% java Gaussian 820 1019 209
0.171
% java Gaussian 1500 1019 209
0.989
% java Gaussian 1500 1025 231
0.980
```

使用静态方法来组织代码 静态方法实现了在输入值的基础上计算输出值的过程，因此，除了用于实现数学函数之外，作为在计算任务中组织控制流的通用技术，也是很重要的。这样做体现了一个非常重要的原则，也是程序员的重要指导原则，即，在编程时，要尽可能地将程序明确地分割成相互独立的子任务，然后分别实现。

对于表达计算任务，使用函数是非常自然的。实际上，我们在1.1节接触的第一个Java程序（“Java程序的直观图”），就是把它当成一个函数来看待的：从那里开始，我们就将Java程序视作将命令行参数转换为输出字符串的函数。在一个计算任务的不同层面，我们都可以按照这样的方式来分析问题（即在不同层面将计算的子任务简单地看成处理输入数据并产生输出数据的函数——译者注）。而且，在通常情况下，我们也自然地把大段的程序以功能为单位进行分割，以函数的方式来表示，而不是把它们简单地记作Java赋值、条件和循环语句的序列。有了定义函数的能力，我们可以通过在适当的时候定义函数来更好地组织程序。

例如, Coupon (程序 2.1.3) 是 CouponCollector (程序 1.4.2) 的一个版本, 它更明确地分离了计算的各个组成部分。例如在程序 1.4.2 中, 你要确定三个独立的任务:

- 给定 n , 计算随机卡券的值。
- 给定 n , 做卡券收集实验。
- 从命令行获取 n , 然后计算并打印结果。

Coupon 重新排列 CouponCollector 中的代码, 可以清晰地反映出这三个函数如何配合并最终实现了计算任务。这样组织代码之后, 我们可以改变 getCoupon() (例如, 我们可能希望从不同的分布中抽取随机数) 或 main() (例如, 我们可能需要多次输入或运行多个实验), 而不必担心 collectCoupons() 会有任何改变。

使用静态方法可以将实验的不同组成部分相互隔离或者封装 (encapsulate)。通常, 程序由许多不同的部分组成, 将其明确地划分为不同的静态方法的做法更有利。在我们再看到其他几个例子之后, 我们将进一步详细讨论这些益处。其实这样做的好处非常容易理解, 在程序中通过将它分解成函数来更好地表达计算, 就像在文章中把它分成几段能更好地表达一个想法一样。在编程时, 要尽可能地将程序明确地分割成相互独立的子任务, 然后分别实现。

程序2.1.3 模拟卡券收集 (二)

```
public class Coupon
{
    public static int getCoupon(int n)
    { // 返回一个0到n-1之间的随机整数
      return (int) (Math.random() * n);
    }

    public static int collectCoupons(int n)
    { // 收集卡券, 直到每张都有
      // 并返回收集到的卡券的数量
      boolean[] isCollected = new boolean[n];
      int count = 0, distinct = 0;
      while (distinct < n)
      {
          int r = getCoupon(n);
          count++;
          if (!isCollected[r])
              distinct++;
          isCollected[r] = true;
      }
      return count;
    }

    public static void main(String[] args)
    { // 收集到了n张不同的卡券
      int n = Integer.parseInt(args[0]);
      int count = collectCoupons(n);
      StdOut.println(count);
    }
}
```

n	卡券的值 (0到 $n-1$ 之间)
isCollected[]	卡券是否已经被收集到?
count	收集到的卡券的数量
distinct	收集到不同卡券的数量
r	随机的卡券

这是程序1.4.2的另一个实现版本。这个版本用于说明在静态方法中封装计算的代码样式。此代码与CouponCollector的效果相同, 但组织结构更好, 它将代码分成三个组成部分: 在0和 $n-1$ 之间生成一个随机整数、运行一个卡券收集实验, 以及管理输入输出。

```
% java Coupon 1000
6522
% java Coupon 1000
6481
```

```
% java Coupon 10000
105798
% java Coupon 1000000
12783771
```

传递参数和返回值 接下来，我们将研究 Java 中参数传递给函数和取得返回值的具体机制。这些机制在概念上是非常简单的，但值得花时间去充分理解，因为这套机制的作用是深刻的。了解参数传递和返回值机制是学习任何新的编程语言的关键。

按值传递。你可以在函数本体的任意位置代码中使用参数变量，方法与使用局部变量的方式相同。参数变量和局部变量之间的唯一区别是：Java 会计算调用者传递的参数表达式的值，然后用计算的结果来为参数变量初始化。此方法称为按值传递（pass by value）。该方法使用其参数的值，而不是参数本身。这种方法的一个结果是，改变参数变量的值对此静态方法中的调用代码没有影响（为了清楚起见，我们不会在本书的代码中更改参数变量）。在某些编程环境中，有一种称为通过引用传递（pass by reference）的替代方法，该方法可以直接修改调用代码的参数。

静态方法可以将数组作为参数，或者将数组返回给调用者。这个功能是 Java 面向对象的一个特例，这也是第 3 章的主题之一。因为基本的机制很容易理解和使用，我们在本节中开始学习这一点，使得我们可以使用数组处理大量数据，从而为许多问题提供编程解决方案。

数组作为参数。当静态方法将数组作为参数时，它可以实现对同一类型的任意数量的值进行操作。例如，下面的静态方法计算双精度数组的平均值：

```
public static double mean(double[] a)
{
    double sum = 0.0;
    for (int i = 0; i < a.length; i++)
        sum += a[i];
    return sum / a.length;
}
```

207

实际上，从第一个程序开始，我们就一直在使用数组作为参数。代码

```
public static void main(String[] args)
```

这一句将 main() 定义为静态方法，它将字符串数组作为参数并不返回任何值。按照约定，Java 系统会将你在 Java 命令中程序名后键入的字符串收集到一个数组中，并将该数组作为参数调用 main()（大多数程序员使用 arg 作为参数变量的名称，虽然任何名称都是可以的）。在 main() 中，我们可以像处理任何其他数组一样操作该数组。

数组的副作用。通常情况下，以数组作为参数的静态方法的目的是产生副作用（更改数组元素的值）。这种方法的一个典型示例是在给定数组中，交换两个给定索引位置的值。我们可以修改 1.4 节开始给出的代码：

```
public static void exchange(String[] a, int i, int j)
{
    String temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

这种实现方式源于 Java 的数组表示形式。exchange() 中的参数变量是对数组的引用，而不是数组值的副本：当将数组作为参数传递给方法时，该方法有机会将值重新分配给该数组中的元素。采用数组参数并产生副作用的静态方法的第二个典型示例是基于 1.4 节中代码而来的 shuffle 函数，这个函数用于随机地打乱数组中的值（其中使用了本节前面提到的 exchange() 和 uniform() 方法）：

```

public static void shuffle(String[] a)
{
    int n = a.length;
    for (int i = 0; i < n; i++)
        exchange(a, i, i + uniform(n-i));
}

```

208

查找数组中的最大值	<pre> public static double max(double[] a) { double max = Double.NEGATIVE_INFINITY; for (int i = 0; i < a.length; i++) if (a[i] > max) max = a[i]; return max; } </pre>
点积	<pre> public static double dot(double[] a, double[] b) { double sum = 0.0; for (int i = 0; i < a.length; i++) sum += a[i] * b[i]; return sum; } </pre>
交换数组中的两个元素的值	<pre> public static void exchange(String[] a, int i, int j) { String temp = a[i]; a[i] = a[j]; a[j] = temp; } </pre>
打印一维数组 (及其长度)	<pre> public static void print(double[] a) { StdOut.println(a.length); for (int i = 0; i < a.length; i++) StdOut.println(a[i]); } </pre>
以行为主顺序读取 二维数组(带维度)	<pre> public static double[][] readDouble2D() { int m = StdIn.readInt(); int n = StdIn.readInt(); double[][] a = new double[m][n]; for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) a[i][j] = StdIn.readDouble(); return a; } </pre>

209

使用数组做参数或返回值的函数的典型代码实现

类似地，我们将在 4.2 节讨论如何对数组进行排序（重新排列它们的值以使它们按顺序排列）。所有这些例子都是在强调以下基本的事实，在 Java 的数组传递机制中，对于数组的引用实现的是按值传递，而对于数组元素本身实现的是按引用传递。与基本类型参数不同，函数对数组元素所做的更改能够反映在调用者程序中。以数组作为参数的函数不能更改数组本身——数组的内存位置、长度和类型，均与创建数组时相同，但函数可以修改数组中元素的值。

数组作为返回值。如果一个方法对参数传递的数组进行了排序、重组，或以其他方式修改，不必返回对该数组的引用，因为它正在更改调用者数组的元素而不是副本。但是在很多情况下，静态方法需要提供一个数组作为返回值，特别是某些静态方法的主要目的是返回多个相同类型的值给调用端。例如，以下静态方法创建并返回 StdAudio 使用的数组（见程序 1.5.7）：它包含了以给定频率（单位为赫兹）的正弦波和给定的持续时间（单位为秒）来采样的值，标准为每秒 44 100 个样本。


```

public static double[] tone(double hz, double t)
{
    int SAMPLING_RATE = 44100;
    int n = (int) (SAMPLING_RATE * t);
    double[] a = new double[n+1];
    for (int i = 0; i <= n; i++)
        a[i] = Math.sin(2 * Math.PI * i * hz / SAMPLING_RATE);
    return a;
}

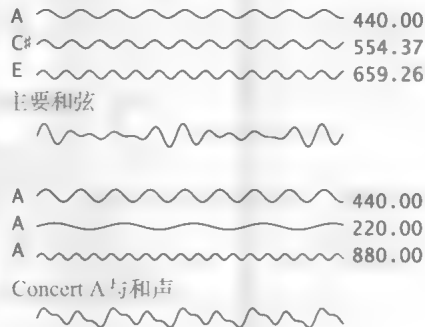
```

在此代码中，返回的数组长度取决于持续时间：如果给定的持续时间为 t ，则数组的长度大约为 $44\,100\,t$ 。通过类似静态方法，我们可以编写将声波视为单个实体的代码（包含采样值的数组），如程序 2.1.4 中所见。

210

示例：声波叠加 如 1.5 节中讨论的，我们研究的简单音频模型需要进行“润色”，以产生类似于乐器发出的声音。润色的方法有很多，我们可以用函数来实现一种。我们系统地使用静态方法创建的声波，远比 1.5 节介绍的简易正弦波要复杂得多。这是一个复杂而有趣的计算问题，为了有效地描述如何解决这个问题，我们想象一个程序，它必须与 PlayThatTune（程序 1.5.7）具有相同的功能，但可以分别增加一个高八度和一个低八度的和弦，以创建更真实的声音效果。

和弦与和声。类似 Concert A 的音符声调单一，听起来都不是很悦耳，因为我们习惯听到的声音含有许多其他成分。比如吉他琴弦的声音是由乐器的木制部分、你所在的房间的墙壁等各个因素相互影响的结果。你也可以想象修改基本正弦波的效果，比如，大部分乐器都能产生和声（同一音符不同的八度，而不是加大音量），或者你可以演奏和弦（多个音符同时演奏）。要结合多个声音，我们就要使用叠加（superposition）：将波进行简单叠加并进行缩放，以确保所有值保持在 -1 和 $+1$ 之间。事实证明，当我们以这种方式叠加不同频率的正弦波时，我们可以得到任意复杂的波。事实上，19 世纪数学的伟大成就之一就是任何平滑的周期函数可以表示为正弦和余弦波的和，这就是傅里叶级数（Fourier series）。对应这个数学概念，则表示我们可以用乐器或我们的声带在极大范围内创建声音，所有声音都由各种振荡曲线的组成构成。声音对应曲线，曲线对应声音，我们可以创建出任意复杂的叠加曲线。



通过叠加波来合成声音

211

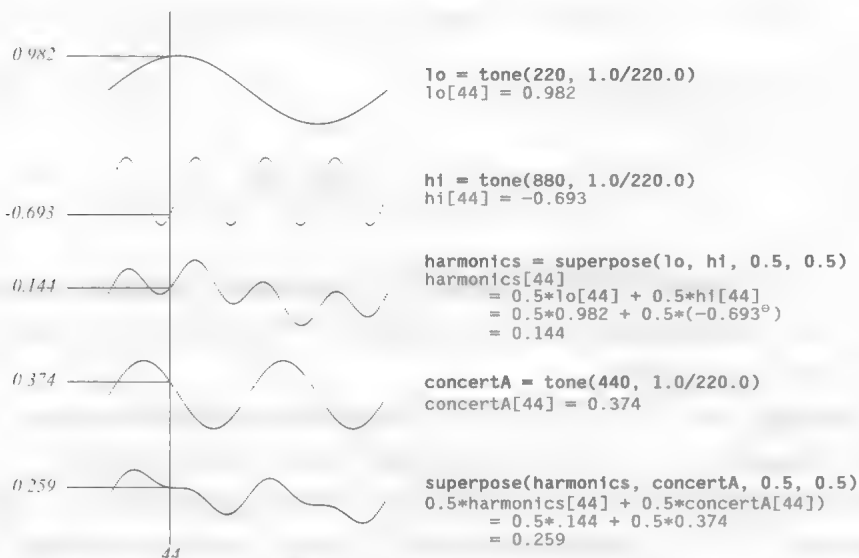
加权叠加。为了在计算机中存储和表示声波，我们在相同的采样点上对声波进行采样，然后将这些采样得到的数字组合成为数组来表示声波。按照这样的方式，声波叠加是很容易实现的：我们把每个采样点的值相加，然后按照比例重新缩放，即可达到叠加的效果。为了更好地控制，我们为要叠加的两个波各指定一个相对权重，权重应该是正数且总和为 1。例如，如果我们想要第一个声音的效果是第二个声音的 3 倍，第一个声音的权重将分配为 0.75，第二个的权重为 0.25。现在，假设一个波存储在数组 $a[]$ 中，相对权重为 awt ；另一个存储在数组 $b[]$ 中，相对权重为 bwt ，我们计算它们的加权求和的代码如下：

```

double[] c = new double[a.length];
for (int i = 0; i < a.length; i++)
    c[i] = a[i]*awt + b[i]*bwt;

```

权重必须为正、总和为 1 确保了叠加后的波的所有值都保持在 -1 和 $+1$ 之间。



向 Concert A 添加和声 (1/220 秒, 44 100 个样本 / 秒)

程序2.1.4 按调演奏 (改进版)

```
public class PlayThatTuneDeluxe
{
    public static double[] superpose(double[] a, double[] b,
                                     double awt, double bwt)
    {
        // 将a和b按照权重叠加
        double[] c = new double[a.length];
        for (int i = 0; i < a.length; i++)
            c[i] = a[i]*awt + b[i]*bwt;
        return c;
    }

    public static double[] tone(double hz, double t)
    {
        // 如正文所示
    }

    public static double[] note(int pitch, double t)
    {
        // 演奏一个给定音高的音符, 并伴随和声
        double hz = 440.0 * Math.pow(2, pitch / 12.0);
        double[] a = tone(hz, t);
        double[] hi = tone(2*hz, t);
        double[] lo = tone(hz/2, t);
        double[] h = superpose(hi, lo, 0.5, 0.5);
        return superpose(a, h, 0.5, 0.5);
    }

    public static void main(String[] args)
    {
        // 读取并演奏一首曲子, 伴随和声
        while (!StdIn.isEmpty())
        {
            // 读取并演奏一个音符, 并伴随和声
            int pitch = StdIn.readInt();
            double duration = StdIn.readDouble();
            double[] a = note(pitch, duration);
            StdAudio.play(a);
        }
    }
}
```

hz 频率
a[] 纯音
hi[] 较高的和声
lo[] 较低的和声
h[] 添加和声后的声音

该代码使用静态方法来润色程序1.5.7生成的声音, 从而产生了比纯音更逼真的和声。

```
% more elise.txt
7 0.25
6 0.25
7 0.25
6 0.25
...
```

```
% java PlayThatTuneDeluxe < elise.txt
```

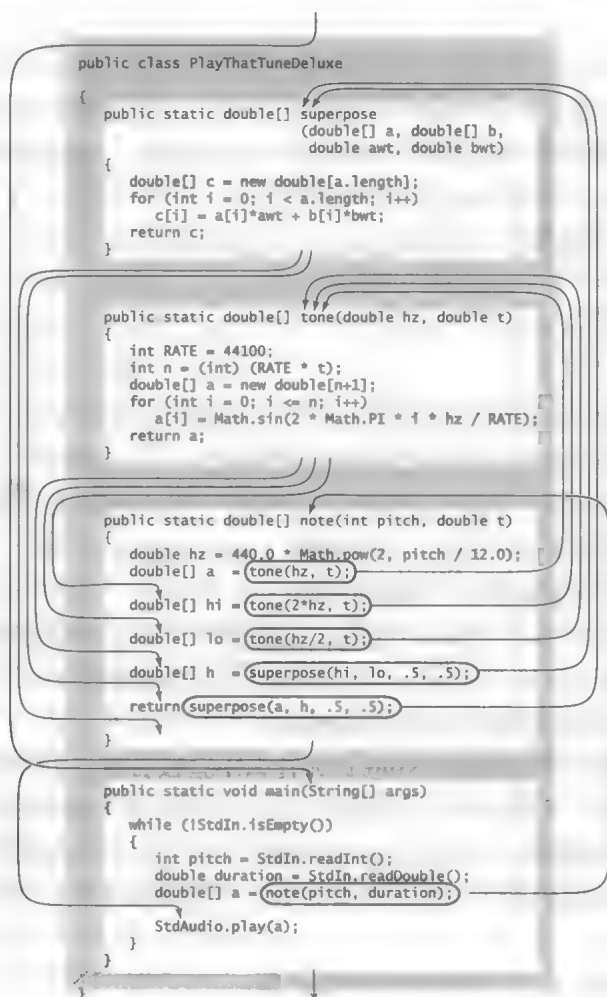


程序 2.1.4 实现了以上这些设计和概念，并生成比程序 1.5.7 更逼真的声音。为了实现这一点，它使用函数将计算任务分为四个部分：

- 给定频率和持续时间，创建一段纯音。
- 给出两个声波和相对权重，将它们叠加起来。
- 给定音高和持续时间，创建一个带和声的音符。
- 从标准输入读取一系列音高及其持续时间，在音频输出上播放它们。

这些子任务中，每一个都可以实现为一个函数，然后这些函数相互依赖以完成整个计算任务。每个功能都有明确的定义，并且实现起来也很容易。所有这些函数（包括 `StdAudio`）都是用一系列浮点数组成的数组来表示声音（以每秒 44 100 个样本进行采样）。

到目前为止，函数的使用都大大简化了我们的编程任务。例如，程序 2.1.1~2.1.3 中的控制流很简单，每个函数只在代码中的一个位置调用。相比之下，`PlayThatTuneDeluxe`（程序 2.1.4）更能说明定义函数来组织计算的有效性，因为这个程序中的函数被多次调用。例如，函数 `note()` 两次调用函数 `tone()`、三次调用函数 `superpose()`。如果不使用函数，我们将需要把 `tone()` 和 `superpose()` 的代码在程序中复制很多次；在使用函数之后，我们可以按功能将这些代码片段封装，然后在编写程序的过程中更加关注于应用功能的设计。同循环很类似，函数也有一个简单而意义深远的效果：一个语句序列（在函数中定义）在程序执行期间可以被多次执行——每当 `main()` 的控制流中调用到这个函数时就会被执行一次。



几个静态方法的控制流

函数（静态方法）很重要，使用它们，我们可以在程序中扩展 Java 语言。对于那些已经实现好并且充分调试的函数，如 `sum()`、`pdf()`、`cdf()`、`mean()`、`abs()`、`exchange()`、`shuffle()`、`isPrime()` 和 `tone()`，我们可以在其他程序中直接使用它们，就好像它们是 Java 中原生支持的函数和功能一样。这样的设计在很大程度上提高了灵活性，为我们开辟了一个全新的编程世界。之前，你可以将 Java 程序看作一系列语句的组合；现在，你需要将 Java 程序视为一组可以互相调用的静态方法。你已经习惯使用的语句到语句的控制流仍然存在于静态方法中，只是程序具有了更高级别的控制流，这个控制流就是静态方法调用和返回。这使得你能够根据应用所要求的操作来思考程序的设计，而不仅仅是对内置于 Java 中的基本类型的简单算术运算。

在编程时，要尽可能地将程序明确地分割成相互独立的子任务，然后分别实现。本节中的例子（以及本书后面的程序）清楚地说明了这一点。使用静态方法，我们可以：

- 将较长的语句序列分成独立的部分。
- 重复使用代码，而无须复制代码。
- 使用更高层次的概念来设计程序（如声波）。

这将生成比仅由 Java 赋值、条件和循环语句组成的长程序更易于理解、维护和调试的代码。在下一节中，我们将讨论如何使用其他程序中定义的静态方法，这会将我们带到一个更高的编程级别。

[215]

问答环节

问：如果我在定义静态方法时省略了关键字 `static`，会发生什么？

答：通常，回答这样的问题最好的方法是自己试一试，看看会发生什么。下面是在 `Harmonic` 的 `harmonic()` 中省略 `static` 修饰符的结果：

```
Harmonic.java:15: error: non-static method harmonic(int)
cannot be referenced from a static context
    double value = harmonic(arg);
                   ^
1 error
```

非静态方法与静态方法不同。你将在第 3 章中了解到它们的差异。

问：如果我在 `return` 语句后编写代码会发生什么？

答：一旦到达返回语句，控制立即返回到调用方，因此返回语句后的任何代码都是无用的。Java 将此情况标识为编译时错误，将会报告：`unreachable code`。

问：如果我不使用 `return` 语句，会发生什么？

答：如果返回类型是 `void`，则没有问题。在这种情况下，控制流会在完成最后一个语句之后返回调用方。如果返回类型不是 `void`，且有任何代码执行路径不是以 `return` 语句结束，则 Java 将报告编译错误：`missing return statement`。

问：为什么我需要使用返回类型 `void`？为什么不直接省略返回类型？

答：Java 语言的设计者就是这么要求的，在没有返回值的时候我们必须写上 `void`。试图揣测编程语言设计者的思路，是成长为一个编程语言设计者迈出的第一步。

问：我可以不使用 `return` 语句从返回类型为 `void` 的函数返回吗？如果是，我应该使用哪个返回值？

[216]

答：可以。使用 `return` 语句返回；没有返回值。

问：程序的副作用和将数组作为参数传递的概念令人费解。这部分真的那么重要吗？

答：是的。在大型系统中，正确控制程序的副作用是程序员最重要的任务之一。花时间来确保你了解按值传递（当参数是基本类型时）和按引用传递（当参数是数组时）之间的区别一定是值得的。同样的机制也用于所有其他类型的数据。你将在第 3 章中再次用到这些知识，并且更加复杂。

问：那么，为什么不将所有的参数都设计为按值传递（包括数组）来消除程序的副作用呢？

答：想象一个巨大的数组，比方说，包含几百万个元素。将所有这些值复制到一个静态方法中，而这个方法仅用于交换其中两个元素的值，是否有意义？因此，大多数编程语言在实现将数组传递给函数时，都选择不创建数组元素的副本——Matlab 是一个例外。

问：Java 计算方法调用的顺序是什么？

答：不管运算符的优先级和结合性是什么，Java 总是从左向右计算子表达式（包括方法调用）和参数列表。例如，在计算表达式

$$f1()+f2() * f3(f4(), f5())$$

时，Java 的调用顺序是 $f1()$ 、 $f2()$ 、 $f4()$ 、 $f5()$ 和 $f3()$ 。这对那些能产生副作用的方法至关重要。作为一种良好的编程风格，我们应该避免编写依赖于计算顺序的代码。

217

练习

- 2.1.1 编写一个静态方法 `max3()`，它需要输入 3 个 `int` 参数并返回最大值。添加一个重载函数，为 3 个 `double` 值做同样的事情。
- 2.1.2 编写一个静态方法 `odd()`，它需要输入 3 个 `boolean` 参数，如果参数值中有奇数个为 `true`（即 1 个或 3 个），返回 `true`，否则为 `false`。
- 2.1.3 编写一个静态方法 `majority()`，需要输入 3 个 `boolean` 参数，如果至少有两个参数值为 `true`，则返回 `true`，否则为 `false`。不要使用 `if` 语句。
- 2.1.4 编写一个静态方法 `eq()`，它将两个 `int` 数组作为参数，如果数组具有相同的长度且所有对应的元素对相等，则返回 `true`，否则为 `false`。
- 2.1.5 编写一个静态方法 `areTriangular()`，它需要输入 3 个 `double` 型参数，如果它们可以是三角形的边（任意一边小于其余两边之和），则返回 `true`。请参阅练习 1.2.15。
- 2.1.6 编写一个静态方法 `sigmoid()`，它需要输入一个 `double` 参数 x 并返回从公式 $1/(1+e^{-x})$ 获得的 `double` 值。
- 2.1.7 编写一个静态方法 `sqrt()`，它需要输入一个 `double` 参数并返回该数字的平方根。使用牛顿方法（参见程序 1.3.6）来计算结果。
- 2.1.8 给出 Java 函数“Harmonic 3 5”的调用轨迹。
- 2.1.9 编写一个静态方法 `lg()`，需要输入一个 `double` 参数 n 并返回 n 的以 2 为底的对数。你可以使用 Java 的数学库 `Math`。
- 2.1.10 编写一个静态方法 `lg()`，它需要输入一个 `int` 参数 n ，并返回不大于 n 的以 2 为底的对数的最大整数。不要使用 `Math` 库。
- 2.1.11 编写一个静态方法 `signum()`，它需要输入一个 `int` 参数 n ：如果 n 小于 0，则返回 -1；如果 n 等于 0，则为 0；如果 n 大于 0，则为 +1。
- 2.1.12 思考下面的静态方法 `duplicate()`。

218

```
public static String duplicate(String s)
{
    String t = s + s;
    return t;
}
```

下面的代码片段执行什么操作？

```
String s = "Hello";
s = duplicate(s);
String t = "Bye";
t = duplicate(duplicate(duplicate(t)));
StdOut.println(s + t);
```

2.1.13 思考下面的静态方法 cube()。

```
public static void cube(int i)
{
    i = i * i * i;
}
```

下面的循环迭代的次数是多少？

```
for (int i = 0; i < 1000; i++)
    cube(i);
```

答案：一定是 1000 次。对 cube() 的调用对调用者端代码没有影响。它更改其局部参数变量 i 的值，但该更改对 for 循环中的 i 没有影响，这是两个不同的变量。如果你使用语句 $i = i * i * i$ 替换对 cube(i) 的调用（可能这就是你正在想的），那么循环迭代 5 次，i 在这 5 次迭代开始时取值分别是 0、1、2、9 和 730。

219

2.1.14 以下校验和公式在银行和信用卡公司广泛使用，用来验证账号的合法性：

$$d_0 + f(d_1) + d_2 + f(d_3) + d_4 + f(d_5) + \cdots = 0 \pmod{10}$$

d_i 是账号数的十进制数的第 i 位， $f(d)$ 是 $2d$ 的十进制数的各位数字总和（例如， $f(7)=5$ ，因为 $2 \times 7=14$ 和 $1+4=5$ ）。例如，17327 是有效的，因为 $1+5+3+4+7=20$ ，是 10 的整数倍。请实现函数 f ，并编写一个程序，接收一个 10 位整数作为命令行参数，输入一个 11 位的有效账号（即符合上面的校验和计算规则的账号——译者注）。账号的第 11 位是一个校验位，由给定的前 10 位数字计算得到。

2.1.15 给定两颗星星，其赤纬 (declination) 和赤经 (right ascension) 位置分别是 (d_1, a_1) 和 (d_2, a_2) ，它们的相对角度计算公式是：

$$2 \arcsin((\sin^2(d/2) + \cos(d_1)\cos(d_2)\sin^2(a/2))^{\frac{1}{2}})$$

其中 a_1 和 a_2 的取值范围是 -180° 和 180° 之间的角度， d_1 和 d_2 的取值范围是 -90° 和 90° 之间的角度， $a=a_2-a_1$ 且 $d=d_2-d_1$ 。编写一个程序，将二星的赤纬和赤经位置作为命令行参数，并打印它们的相对角度。提示：注意需要从角度转换为弧度。

2.1.16 写一个静态方法 scale()，它需要输入一个 double 型数组作为参数，它的副作用是将数组中的每个元素缩放到 0 和 1 之间（每个元素减去最小值，然后除以最小和最大值之间的差）。使用本书表格中定义的 max() 方法，并编写和使用相应的 min() 方法。

2.1.17 编写一个静态方法 reverse()，它需要输入一个字符串数组作为其参数，它会创建一个新数组，并使其中的元素以原来数组元素相反的顺序排列，返回这个新数组（不要更改参数数组中字符串的顺序）。再编写一个静态方法 reverseInplace()，它采用一个字符串数组作为其参数，并反转参数数组中字符串顺序。

220

- 2.1.18 编写一个静态方法 `readBoolean2D()`，它从标准输入读取二维 `boolean` 型矩阵（带有维度），并返回生成的二维数组。
- 2.1.19 编写一个静态方法 `histogram()`，它使用 `int` 型数组 `a[]` 和整数 `m` 作为参数，并返回一个长度为 `m` 的数组，其第 `i` 个元素是 `i` 出现在 `a[]` 中的次数。假设 `a[]` 中的值都在 0 和 `m-1` 之间，则返回数组中的值之和应等于 `a.length`。
- 2.1.20 结合本节和 1.4 节中的代码片段，开发一个程序，该程序需要输入整数命令行参数 `n`，从随机混排的卡券盒中抽取 `n` 手牌，每手 5 张。打印这 `n` 手牌的内容，其中每张牌占一行，打印牌的名称，类似于“`Ace of Clubs`”，每手牌之间以空白行分隔。
- 2.1.21 写一个静态方法 `multiply()`，它需要输入两个参数维度相同的方阵作为参数，求它们的乘积（结果是一个相同维度的矩阵）。加分题：确保第一个矩阵中的列数等于第二个矩阵中的行数。
- 2.1.22 编写一个静态方法 `any()`，它需要输入一个 `boolean` 型数组作为参数，如果数组中的任一元素为 `true`，则返回 `true`，否则返回 `false`。编写一个静态方法 `all()`，它需要输入一个 `boolean` 型数组作为参数，如果数组中的所有元素均为 `true`，则返回 `true`，否则返回 `false`。
- 2.1.23 开发一个版本的 `getCoupon()`，以模拟有一张较为罕见的卡券的情况：随机选择一个 `n` 值，以 $1/(1000n)$ 的概率返回该值，以相同概率返回其他值。加分题：这种变化对卡券收集时所需拿到的卡券平均值有什么影响？
- 2.1.24 修改 `PlayThatTune`，为每个音符叠加与原来的音符相隔两个八度的和声，这两个和声的权重都设置为原一个八度和声的一半。

221

创新练习

- 2.1.25 生日问题。开发一个类，其中包含若干个适当的静态函数，用以研究生日问题（参见练习 1.4.38）。
- 2.1.26 欧拉函数。欧拉函数是数论中的重要函数： $\varphi(n)$ 被定义为小于或等于 n 的数中与 n 互质的数的数目（即与 n 之间除了 1 之外没有其他公因子）。编写一个类并实现一个静态方法，它需要输入一个整数参数 n 并返回 $\varphi(n)$ ，然后编写一个 `main()` 函数，它需要输入一个整型命令行参数，用这个参数调用该方法，并打印结果值。
- 2.1.27 谐波数。编写一个程序 `Harmonic`，其中包含三个静态方法 `harmonic()`、`harmonicSmall()` 和 `harmonicLarge()` 来计算谐波数。`harmonicSmall()` 方法应该只计算累加和（如程序 1.3.5），`harmonicLarge()` 方法使用近似公式 $H_n = \log_e(n) + \gamma + 1/2n - 1/(12n^2) + 1/(120n^4)$ ($\gamma = 0.577215664901532\cdots$ ，称为欧拉常数)，当 $n < 100$ 时，`harmonic()` 方法应该调用 `harmonicSmall()`，否则调用 `harmonicLarge()`。
- 2.1.28 布莱克 - 斯科尔斯期权估价。布莱克 - 斯科尔斯公式为欧洲认购非股息的股票期权提供了理论基础。假定当前股票价格 s ，执行价格 x ，连续复合无风险利率 r ，波动率 σ 和成熟期（年） t 。布莱克 - 斯科尔斯值由公式 $s\Phi(a) - xe^{-rt}\Phi(b)$ 给出，其中 $\Phi(z)$ 是高斯累积分布函数， $a = (\ln(s/x) + (r + \sigma^2/2)t)/(\sigma\sqrt{t})$ ， $b = a - \sigma\sqrt{t}$ 。编写一个程序，从命令行中获取 s 、 r 、 σ 和 t ，并打印布莱克 - 斯科尔斯值。
- 2.1.29 傅里叶峰值。编写一个程序，需要输入命令行参数 n 并绘制函数

$$(\cos(t) + \cos(2t) + \cos(3t) + \cdots + \cos(nt))/n$$

从 -10 到 10（以弧度表示）之间取 500 个相同间隔的 t 样本。令 $n=5$ 和 $n=500$ 时，运行你的程序。注意：你将观察到总和收敛到一个尖峰（除了单个地方有值，其他地方都是 0）。该属性可以用于证明任何平滑函数可以表示为正弦曲线的累加和。

222

- 2.1.30 日历。编写一个程序日历，它需要输入两个整型命令参数 m 和 y ，并打印年度 y 的第 m 月的

日历，如以下示例所示：

```
% java Calendar 2 2009
February 2009
S M Tu W Th F S
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

提示：请参见程序 LeapYear（程序 1.2.4）和练习 1.2.29。

- 2.1.31 霍纳方法。编写一个类 Horner 并实现一个方法 evaluate()，该方法使用浮点数 x 和数组 $p[]$ 作为参数，计算其系数为 $p[]$ 的 x 的多项式的值并返回计算结果，多项式如下：

$$p(x) = p_0 + p_1x + p_2x^2 + \cdots + p_{n-2}x^{n-2} + p_{n-1}x^{n-1}$$

霍纳方法是以上多项式的一种有效计算方法，即按照以下公式完成对上述多项式的计算：

$$p(x) = p_0 + x(p_1 + x(p_2 + \cdots + x(p_{n-2} + xp_{n-1}))) \cdots$$

编写一个测试用客户端静态方法 exp()，该方法使用 evaluate() 来计算 e^x 的近似值，你可以使用泰勒级数扩展 $e^x = 1 + x/1! + x^2/2! + x^3/3! + \cdots$ 的前 n 项来实现计算任务。你的客户端程序应该需要输入命令行参数 x ，并将其结果与 Math.exp(x) 计算结果进行比较

- 2.1.32 和弦。开发一个可以处理和弦（包括和声）的歌曲的 PlayThatTune。开发一种输入格式，允许你为每个和弦指定不同的持续时间，并为和弦中的每个音符指定不同的权重。通过各种和声和和弦的测试文件来测试你的程序，并用它来尝试演奏《致爱丽丝》。

223

- 2.1.33 本福德定律。美国天文学家西蒙·纽科姆（Simon Newcomb）在一本对数表（对数表是在计算机出现之前常用于求对数值的工具书——译者注）中观察到了一个奇怪的现象：开始页面比结尾页面要脏得多。他怀疑科学家在计算过程中，用 1 开始的数字，要比用 8 或者 9 开始的数字多很多，并且推测在一般情况下，以 1 为首位数字的数的出现概率（近 30%）远比起以 9 开始的数字的出现概率（少于 4%）要大。这种现象被称为本福德定律，现在经常用作统计检验。例如，国税局法务会计师依靠它来发现税务欺诈。编写一个程序，将这一过程分解为一组静态方法，从标准输入读入整数序列，并列出 1~9 每个数字作为首位数字的次数。使用你的程序来测试计算机或网络上的一些信息表，看它们是否遵从这一法则。然后，编写一个程序，通过生成从 \$ 1.00 到 \$ 1,000.00 的随机金额，遵从相同的概率分布，看你是否能以此打败国税局（IRS）。

- 2.1.34 二项分布。写一个函数：

```
public static double binomial(int n, int k, double p)
```

用如下公式计算： n 次独立重复的丢硬币试验中出现 k 次正面的概率（假设正面出现的概率是 p ）

$$f(n, k, p) = p^k (1-p)^{n-k} n! / (k!(n-k)!)$$

提示：为了避免溢出，计算 $x = \ln f(n, k, p)$ ，然后返回 e^x 。在 main() 中，从命令行中取 n 和 p ，并检查对于所有 k 的取值（在 0 和 n 之间）的概率总和是否为 1（或近似为 1）。此外，将计算的结果与正态分布的近似值相比较

$$f(n, k, p) \approx \phi(np, np(1-p))$$

（见练习 2.2.1）。

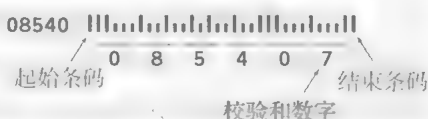
2.1.35 基于二项分布的卡券收集。开发一个新的 `getCoupon()` 版本，使用前面练习中的 `binomial()`，根据 $p=1/2$ 的二项分布返回卡券值。提示：生成一个 0 到 1 之间的随机数 x ，然后返回满足下列条件的最小的 k ：对于所有的 $j < k$ ， $f(n, j, p)$ 之和大于 x 。加分题：在此假设条件下，尝试描述卡券收集函数的行为。

224

2.1.36 邮政条码。美国邮政系统用于邮寄邮件的条形码遵循如下规则：邮政编码（ZIP 码）中的每个十进制数字编码由 3 个半长、2 个整长的条码组成。条形码以整长的条码开始，以整长的条码结束，并包含校验和数字（跟在第 5 位 ZIP 码或 ZIP+4 之后），校验和数字是原始数字之和除以 10 的余数。请定义函数实现以下功能：

- 在 `StdDraw` 上绘制半长或整长的条码。
- 给定一个数字，绘制它的条码。
- 计算校验和数字。

还要实现一个测试客户端，它读取一个 5 位（或 9 位）数字的邮政编码作为命令行参数，并绘制相应的邮政条码。



225

2.2 库和客户程序

迄今为止，你所写的程序都只是存在于单个 `.java` 文件中的 Java 代码。而对于大型程序，将所有代码保存在单个文件中，这种方式存在着很多限制，而且也没有必要这么做。幸运的是，在 Java 中，引用在另一个文件中定义的方法非常容易。Java 的这种能力对我们的编程风格有两个重要的影响。

首先，它使代码复用（code reuse）成为可能。一个程序可以通过引用，来使用已经编写和调试过的代码，而不必复制代码。定义可重复使用的代码的能力，是现代编程的重要组成部分。这相当于扩展 Java——你可以定义和使用你自己的数据操作。

其次，它实现了模块化编程（modular programming）。你不仅可以如 2.1 节所述，将程序分成静态方法，还可以将这些方法保存在不同的文件中，并根据应用程序的需要进行分组。模块化编程非常重要，因为它允许我们独立地开发、编译和调试一个大型程序的每个部分，并将每个完成的代码块放在它自己的文件中以备以后使用，而无须再担心其细节。我们可以开发静态方法库以供其他程序使用，将它们保存在本身的文件中，并且可以在其他程序中使用这些方法。你已经使用过的例子有 Java 的 `Math` 库和我们用于输入/输出的 `Std*` 库。更重要的是，你很快就能发现定义自己的库也非常容易。定义库并且在多个程序中使用它们的能力是我们构建程序来实现复杂任务的能力的关键组成部分。

刚刚在 2.1 节中，我们从把 Java 程序当作一系列语句，转换到了把 Java 程序当作一个包含了一组静态方法（其中一个是 `main()`）的类。在本节，你要做好准备把 Java 程序看作一组类（class），每个类都是由一组方法组成的独立模块。由于每个方法都可以调用另一个类中的方法，所以所有代码都可以互相调用来进行交互，并且组合在一起形成一个复杂的网。通过这种能力，你可以考虑在编程时通过将编程任务分解为可独立实施和测试的类，来降低编程的复杂程度。

226

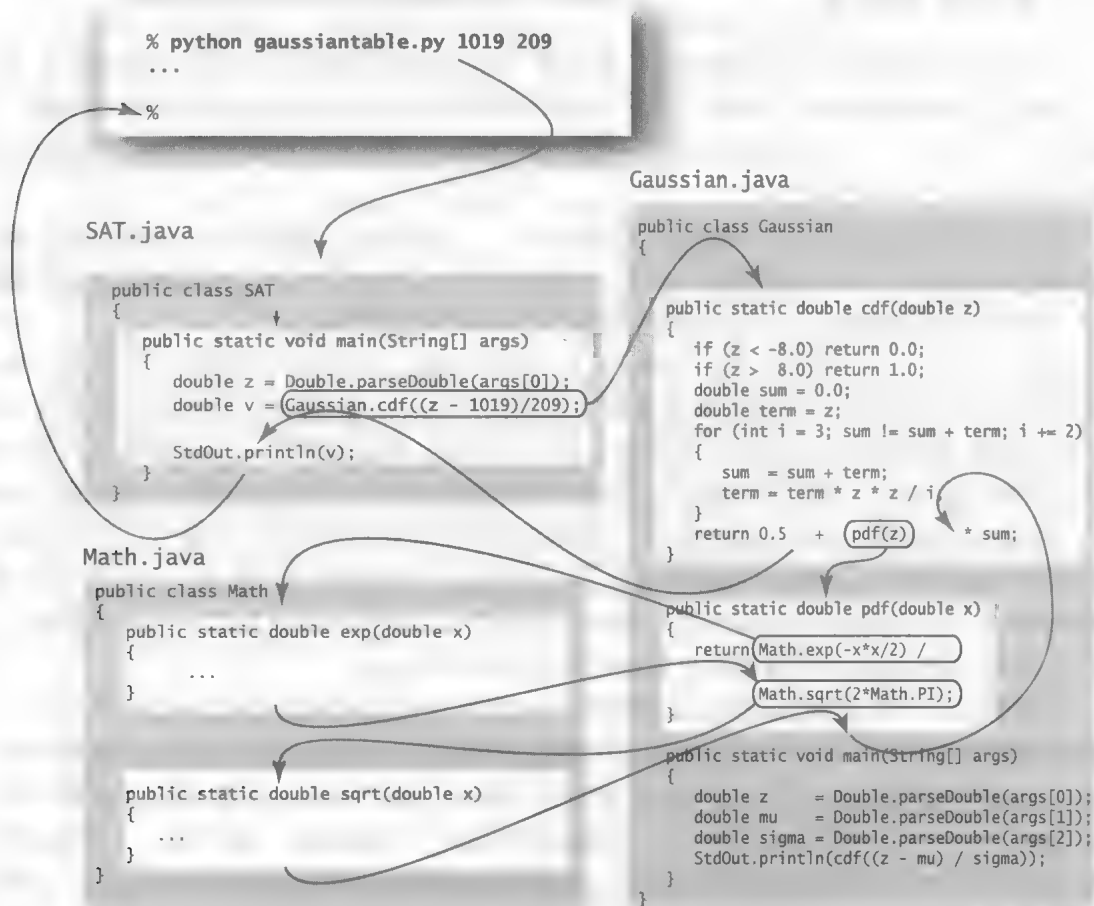
在其他程序中使用静态方法 要在一个类中调用另一个类定义的方法，我们可以使用调用 `Math.sqrt()` 和 `StdOut.println()` 等方法一样的机制：

- 使两个类都可以被 Java 访问（例如，将它们都放在你的计算机中的同一目录中）。

- 想要调用一个方法，需要在这个方法名的前面加上其类名，并用一个点号做分隔符。

举例来说，我们希望编写一个简单的客户端 SAT.java，它从命令行获取 SAT 分数 z ，并打印指定年份中得分小于 z 的学生的百分比（假设这一年的平均分数为 1019，标准差是 209）。要实现这个功能，SAT.java 需要计算 $\Phi((z-1019)/209)$ ，这一步适合调用 Gaussian.java 中的 cdf() 方法实现。我们现在只需要将 Gaussian.java 放在和 SAT.java 相同的目录下，并且在调用 cdf() 时指定好类名。另外，这个目录下的任何其他方法都可以通过类似 Gaussian.pdf() 或 Gaussian.cdf() 这样的形式使用定义在 Gaussian 中的静态方法。在 Java 中，Math 库总是可以被访问，因此任何类都可以调用 Math.sqrt() 和 Math.exp()。文件 Gaussian.java、SAT.java 和 Math.java 中各自实现了一个 Java 类，并且类之间存在交互：SAT 调用了 Gaussian 中的一个方法，这个方法又接着调用 Math 中的两个方法。

定义多个文件，每个文件都是包含多个方法的独立类，这样的编程方式带来的潜在效果是又一次深刻改变了我们的编程风格。通常，我们把这种方法称为模块化编程。我们在一次应用中开发并调试了一些方法，就可以在以后的任何时间调用它们。在这一节中，我们将通过许多例子来帮助你理解并熟悉这个思路。然而，在举例之前，我们需要讨论几个流程的细节。



模块化程序中的控制流程[⊖]

⊖ 请注意图的左上角那里原书中有错误。那一行 python 命令应该是 java SAT1019 209。——译者注

关键字 `public`。自 `HelloWorld` 以来，我们一直在每个静态方法前都标识了一个关键字 `public`，这一修饰语表示该方法可供任何其他可访问该文件的程序使用。你也可以将方法标识为 `private`（还有其他几个类别），但是你现在没有理由这样做。我们将在 3.3 节讨论这些标识的含义。

每个模块都是一个类（`class`）。我们使用术语模块（`module`）来指代我们保存在单个文件中的所有代码。按照惯例，Java 中的每个模块都是一个 Java 类（`class`），保存这个类的代码的文件名应该与类名相同且扩展名是 `.java`。在本章中，每个类只是一系列静态方法（其中一个为 `main()`）。你将在第 3 章中学习更多的 Java 类的常用结构。

`.class` 文件。当你编译程序（通过键入 `javac` 后跟类名称）时，Java 编译器将创建一个以类名命名的文件，后跟 `.class` 扩展名，这个文件用更适合计算机理解的语言存储着你的代码。如果你有一个 `.class` 文件，即使没有相应的 `.java` 文件中的源代码，你也可以在另一个程序中使用该模块的方法（但是如果你发现了一个代码中的 `bug`，则会束手无策！）。

227
228

需要时再编译。当你编译一个程序时，Java 通常会编译所有能够让这个程序正常运行的部分。如果你在 SAT 中调用 `Gaussian.cdf()`，然后当你输入 `javac SAT.java` 时，编译器将检查自上次被编译至今，`Gaussian.java` 是否被修改（通过比较 `Gaussian.java` 上次被修改的时间和 `Gaussian.class` 创建的时间来实现）。如果它被修改了，它也会重新编译 `Gaussian.java`！仔细思考这个方式，你会觉得它是一个非常有用的方案。毕竟如果你在 `Gaussian.java` 中发现了一个 `bug`（然后修复了它），你会希望所有调用这个方法类都使用最新的这个版本。

多个 `main()` 方法。另一个需要注意的微妙问题是：可能不止一个类包含 `main()` 方法。在我们的例子中，SAT 和 `Gaussian` 都有它们自己的 `main()` 方法。如果你回想一下运行程序的规则，你会发现这里不存在混淆：当你输入 `java` 后跟一个类名，Java 将控制转移到这个类定义的 `main()` 方法的机器码上。通常我们在每个类中都定义一个 `main()` 函数用于测试和调试这个类的方法。当我们想要运行 SAT 时，我们输入 `java SAT`；当我们想要调试 `Gaussian` 时，我们输入 `java Gaussian`（加上合适的命令行参数）。

如果你写的每个程序都可以在别的编程任务中发挥作用，很快你就会发现自己有了各种有用的工具。模块化编程使得我们可以把以往开发的任何计算问题的解决方案都看作后续编程的资源，随时可以用在新的计算任务当中。

举例来说，假设你将来在某个应用程序中需要计算 Φ ，为何不直接剪切和粘贴 `Gaussian` 中实现 `cdf()` 的代码？虽然这是可行的，但是会使你有两份同样的代码，从而代码的维护变得困难。如果之后你想修改或者改进这份代码，你将需要在两份代码中做同样的工作。作为替代，你可以直接调用 `Gaussian.cdf()`。我们自己实现的方法和使用自己的方法都会越来越多，所以只维护一份代码是一个值得追求的目标。

从这一点出发，你写每一个程序时，都应该找到合理的方法，把计算工作分成大小便于管理的几个独立部分，并且在实现每个部分的时候，都要做好以后给别人使用它的准备。最常见的是这个别人一般就是你自己，你会感激自己省下了重新开发或者重新调试代码的功夫。

229

库 我们把其中包含的方法主要用于供其他程序使用的模块叫作库（`library`）。Java 中最重要的编程特性之一就是为了方便用户使用，预定义了成千上万个库。我们会在本书中有选择地介绍一些你可能会感兴趣的库，但是我们不会深入 Java 库的设计细节，因为这些库

中的很多是专为有经验的程序员而设计的。我们将把本章的重心放在更重要的用户自定义库 (user-defined libraries) 上, 自定义库就是一些用户自定义的类, 其中包含了一组供其他程序使用的相关方法。任何一个 Java 库都不能包含我们计算所需要的所有库, 因此创建我们自己的方法库是解决复杂编程问题的关键步骤。

客户。我们使用术语客户 (client) 来表示调用指定库方法的程序。当一个类 A 中包含的一个方法调用了另一个类 B 中的方法时, 我们把类 A 叫作类 B 的客户。在我们的例子中, SAT 是 Gaussian 的一个客户。一个类可能有多个客户。举例来说, 所有你写的调用了 Math.sqrt() 或者 Math.random() 的程序都是 Math 的客户。

应用程序编程接口 (API)。通常, 程序员认为在客户和方法的具体实现描述之间存在着契约 (contract)。当客户和实现都是由你实现时, 你在和你自己定下契约, 契约本身非常实用, 因为它为调试提供了额外的帮助。更重要的是, 这种形式促进了代码的复用。实际上, 你已经编写的很多程序基本都是 Std*、Math 和其他内置 Java 类的客户。这得益于那些非正式的契约 (用我们可以理解的语言准确地描述了这些类和方法的功能) 以及对这些方法的签名的确切描述。这些信息总体上被称为应用程序编程接口 (Application Programming Interface, API)。而对于用户自定义库, 这种机制也是有效的。API 使得任何客户可使用库, 而无须关心代码的具体实现, 正如你在使用 Math 和 Std* 时那样。API 设计的指导原则是仅向客户提供他们需要的方法。具有大量方法的 API 在实现程序上可能是负担; 而缺少重要方法的 API 对客户来说则可能不够方便。

实现。我们用术语实现 (implementation) 来描述实现 API 中方法的 Java 代码, 它通常保存在以库名称命名的 .java 文件中。每个 Java 程序都是一些 API 的实现, 脱离了实现的 API 是毫无用处的。当我们试图为 API 提供实现时, 我们的目标是遵守契约, 即保证 API 的函数功能保持不变。在通常情况下, 对于同一个 API 可以有多种实现方法, 因为我们分离了客户代码与实现代码, 因此可以自由地替换代码以改进 API 的实现。

举例来说, 对于我们前面经常用到的高斯分布函数, 这些函数不在 Java 的 Math 库中, 但在实际应用中很重要, 因此我们把它放在一个库中, 这样它们就可以在将来被客户程序访问。以下是对此 API 的精准描述:

客户



API

```
public class Gaussian
    double pdf(double x)       $\phi(x)$ 
    double cdf(double z)      $\Phi(z)$ 
```

定义函数签名
并描述库方法

实现

```
public class Gaussian
{ ...
    public static double pdf(double x)
    { ... }

    public static double cdf(double z)
    { ... }
}
```

实现库方
法的Java代码

库抽象

```
public class Gaussian
```

double	pdf(double x)	$\phi(x)$
double	pdf(double x, double mu, double sigma)	$\phi(x, \mu, \sigma)$
double	cdf(double z)	$\Phi(z)$
double	cdf(double z, double mu, double sigma)	$\Phi(z, \mu, \sigma)$

高斯分布函数的静态方法库的 API

此 API 不仅包括单参数的高斯分布函数（见程序 2.1.2），而且还包括在许多统计应用中出现的三参数版本（客户指定分布的平均值和标准偏差）。三参数高斯分布函数的实现很简单（参见练习 2.2.1）。

230
231

API 应包含多少信息？这是程序员和计算机科学教育工作者中的一个灰色地带，也是一个被激烈辩论的问题。我们可能尝试在 API 中提供尽可能多的信息，但是（与任何契约一样！），我们可以有效地包含的信息量是有限的。在本书中，我们坚持一个与我们的指导性设计原则相符的原则：仅仅向客户程序员提供他们所需要的信息。相比于提供有关实现的详细信息方案，这样做可以给我们更强的灵活性。事实上，任何额外的信息相当于隐蔽地扩展了契约，这是不必要的。许多程序员陷入分析实现代码的坏习惯中，试图了解它的作用。这样做可能导致客户代码依赖于某些未在 API 中明确指定的行为，一旦 API 的实现发生了改变，程序可能就无法正常工作。API 的实现的更迭可能比你想象的更频繁。例如，每个新版本的 Java 都包含许多新的库函数实现。

通常情况下，实现是早于 API 出现的。你可能首先开发了一个能用的模块并完成了一个计算任务，随后你在新的编程中又想再次使用它，那么你就在新的程序中直接使用了这个模块的方法。在这种情况下，找到一个合适的时机为程序仔细地设计并阐述 API 是一个明智的决定。因为设计这些方法的初衷可能没有考虑到会被复用，所以这时使用 API 来完成代码的可复用设计是非常重要的（就像我们对 Gaussian 程序做的工作一样）。

本节的其余部分专门介绍了库和客户的几个例子。我们考虑这些库的目的是双重的。首先，当你开发越来越多、越来越复杂的程序时，这些库将为你提供更丰富的编程资源。其次，当你开始开发供自己使用的库时，它们可以成为你学习的例子。

随机数 我们已经编写了几个使用 Math.random() 的程序，但是我们的代码经常需要再做一些额外的工作，将 Math.random() 提供的 0 到 1 之间的随机双浮点数值转换为我们想要使用的随机数的类型（比如随机布尔值或特定范围内的随机整型值）。为了有效地复用我们代码实现的这一部分工作，我们将从现在开始使用程序 2.2.1 中实现的 StdRandom 库。StdRandom 使用重载机制实现服从各种分布的随机数生成。你可以按照使用我们自定义的标准 I/O 库相同的方式使用它们中的任何一个（参见 2.1 节“问答环节”中的第一问）像往常一样，我们使用 API 总结了 StdRandom 库中的方法：

232

public class StdRandom	
void setSeed(long seed)	设置种子以获得可重现的结果
int uniform(int n)	0和n-1之间的整数
double uniform(double lo, double hi)	lo和hi之间的浮点数
boolean bernoulli(double p)	true的概率为p, false的概率为1-p
double gaussian()	高斯分布, 均值0, 标准差1
double gaussian(double mu, double sigma)	高斯分布, 均值mu, 标准差sigma
int discrete(double[] p)	以p[i]的概率生成i
void shuffle(double[] a)	随机混排数组a[]中的元素

随机数的静态方法库的 API

这些方法对我们来说十分熟悉，因此 API 中的简短描述足以说明它们所做的工作。我们把这些使用 Math.random() 来生成各种类型的随机数的方法集中在一个文件（StdRandom.

java) 中, 利用这个文件来完成所有的随机数生成任务 (并尽可能地复用该文件中的代码), 而不是把使用这些方法的代码散播到每个程序中。每个使用这些方法的程序都比直接调用 `Math.random()` 的代码更清晰, 因为 `StdRandom` 中提供的方法已经在 API 中清楚地表达了使用 `Math.random()` 实现的功能。

API 设计。我们对传递给 `StdRandom` 中每个方法的值做出某些假设。例如, 我们假设客户仅对正整数 n 调用 `uniform(n)`, 对于 0 和 1 之间的 p 调用 `bernoulli(p)`, 而仅对于元素在 0 和 1 之间并且和为 1 的数组调用 `discrete()`。所有这些假设都是客户与实现之间契约的一部分。我们设计库时, 尽可能地使契约清晰明确, 避免陷入实现细节的泥潭。与编程中的许多任务一样, 良好的 API 设计通常是多次尝试和迭代后的结果。我们总是特别注意 API 的设计, 因为当我们更改 API 时, 我们可能需要更改所有客户代码和所有实现。我们的目标是把“阐明客户如何使用 API”与“API 中的实际代码”剥离开来。这种做法使我们能够更灵活地修改代码, 也可以随时使用更高效或更准确的 API 实现。

233

程序2.2.1 随机数库

```
public class StdRandom
{
    public static int uniform(int n)
    { return (int) (Math.random() * n); }

    public static double uniform(double lo, double hi)
    { return lo + Math.random() * (hi - lo); }

    public static boolean bernoulli(double p)
    { return Math.random() < p; }

    public static double gaussian()
    { /* 见练习2.2.17 */ }

    public static double gaussian(double mu, double sigma)
    { return mu + sigma * gaussian(); }

    public static int discrete(double[] probabilities)
    { /* 见程序1.6.2 */ }

    public static void shuffle(double[] a)
    { /* 见练习2.2.4 */ }

    public static void main(String[] args)
    { /* 见正文 */ }
}
```

这个库中的方法计算了多种类型的随机数: 小于给定值的随机非负整数、在给定范围内均匀分布、随机的二进制位 (伯努利分布)、标准高斯分布、可以设定均值和标准差的高斯分布, 以及给定的某个离散分布。

```
% java StdRandom 5
90 26.36076 false 8.79269 0
13 18.02210 false 9.03992 1
58 56.41176 true 8.80501 0
29 16.68454 false 8.90827 0
85 86.24712 true 8.95228 0
```

234

单元测试。即便我们在实现 `StdRandom` 时没有涉及任何特定的客户, 但是包含一个测试客户 (test client) `main()` 的程序还是很有必要的, 尽管在客户类使用库时不使用它, 但是

在调试和测试库中的方法时会用到。无论何时创建库，都应该包含一个用于单元测试和调试的 `main()` 方法。如果编写一个正确的单元测试本身可能是一个重大的编程挑战（例如，如何测试 `StdRandom` 中的方法产生的数字与真正随机数具有相同特征，专家们仍在讨论）。但至少应该包括一个可以实现如下功能的 `main()` 方法：

- 运行所有代码。
- 在一定程度上验证代码的工作是否正确。
- 从命令行获取参数以便进行更多的测试。

然后，当你想更广泛地使用库时，你应该优化 `main()` 方法来进行更详尽的测试。例如，我们可以从 `StdRandom` 的以下代码开始（以 `shuffle()` 的测试作为练习）：

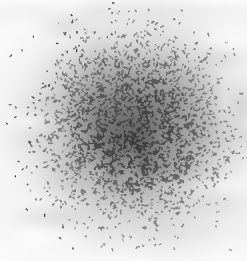
```
public static void main(String[] args)
{
    int n = Integer.parseInt(args[0]);
    double[] probabilities = { 0.5, 0.3, 0.1, 0.1 };
    for (int i = 0; i < n; i++)
    {
        StdOut.printf(" %2d ", uniform(100));
        StdOut.printf("%8.5f ", uniform(10.0, 99.0));
        StdOut.printf("%5b ", bernoulli(0.5));
        StdOut.printf("%7.5f ", gaussian(9.0, 0.2));
        StdOut.printf("%2d ", discrete(probabilities));
        StdOut.println();
    }
}
```

当我们将这个代码包含在 `StdRandom.java` 中并且在程序 2.2.1 中调用这个方法时，输出必然如下：第一列中的整数也可能是从 0 到 99 的任何值；第二列中的数字可能在 10.0 和 99.0 之间均匀分布；第三列中约一半的值为真；第四列的数字平均值约为 9.0，且不可能偏离太多；最后一列约为 50%0s、30%1s、10%2s 和 10%3s。若某列结果不准确，我们可以键入 `java StdRandom 10` 或 `100` 来查看更多结果。在这个例子中，我们可以（并应该）在单独的客户程序中进行更广泛的测试，以检查这些数字是否与服从指定分布的真正随机数具有相同的特性（参见练习 2.2.3）。一个有效的方法是使用 `StdDraw` 来编写测试客户程序，因为数据可视化可以快速识别程序的表现是否符合预期。例如，在程序中生成大量的点，其 x 和 y 坐标都是从各种分布中随机抽取，这些点组成的图像通常能够给出关于这个分布的重要属性的直观感受。除此之外，随机数生成代码中任何的错误都可很容易地在这样的图像中显示出来。

235

压力测试。被广泛使用的库，如 `StdRandom` 也应该进行压力测试（stress test），通过测试我们可以确保当客户不遵守契约或做出一些未明确说明的操作时，它们不会崩溃。Java 自带的库已经通过了这样的压力测试，这需要仔细检查每一行代码，并观察某些情况是否可能导致问题。如果数组元素的和不等于 1，则 `discrete()` 函数会怎么办？如果参数是长度为 0 的数组呢？如果双参数的 `uniform()` 的两个参数中，其中一个或两个均是 NaN 或者无穷，那么应该怎么做？你可以想到的任何问题都是合理的。这种情况有时被称为边界情况（corner cases）。你肯定会遇到对边界情况非常在意的老师或主管。作为经验之谈，大多数程序员应该尽早学会找到边界情况并处理好它们的方法，以避免在测试和调试过程中产生太多的不愉快。再次强调，合理的做法是将压力测试作为一个单独的客户实施。

```
public class RandomPoints
{
    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        for (int i = 0; i < n; i++)
        {
            double x = StdRandom.gaussian(.5, .2);
            double y = StdRandom.gaussian(.5, .2);
            StdDraw.point(x, y);
        }
    }
}
```



236

StdRandom 的一个测试用客户程序

数组的输入和输出 我们已经看到并将继续看到许多例子，我们希望将数据保存在数组中进行处理。因此，构建一个库来完善 StdIn 和 StdOut 是很实用的，它应该提供一些静态方法，用于从标准输入读取基本类型的数组，或者将数组打印到标准输出。以下 API 提供了这些方法：

```
public class StdArrayIO
```

<code>double[]</code>	<code>readDouble1D()</code>	读取一维浮点数数组
<code>double[][]</code>	<code>readDouble2D()</code>	读取二维浮点数数组
<code>void</code>	<code>print(double[] a)</code>	输出一维浮点数数组
<code>void</code>	<code>print(double[][] a)</code>	输出二维浮点数数组

注1：以1D结尾的函数，其格式是1个整数n后跟n个值

注2：以2D结尾的函数，其格式是2个整数m和n，后面跟着m×n个值，以行排序

注3：API中还包括基于int和boolean的相同方法

数组输入输出的静态方法库的 API

上述表格底部的前两个注解提示我们，需要确定一个文件格式（file format）。为了简单和协调，我们假设出现在标准输入中的所有值都会首先标明维度，然后按照上述说明中期望的顺序排列。`read*()` 系列方法期望以此格式作为输入；`print()` 方法以此格式生成输出。表底部的第三个注解表明，StdArrayIO 实际上包含 12 个方法，int 型、double 型和 boolean 型各 4 个。`print()` 方法是重载的（它们都具有相同的名称 `print()`，但参数类型不同），但是 `read*()` 系列方法需要通过添加类型名称（大写，类似于 StdIn 中的格式），后面跟着 1D 或 2D 的形式来生成不同的名称。

通过 1.4 节和 2.1 节中的处理数组代码来实现这些方法是直观的，如 StdArrayIO（程序 2.2.2）所示。将所有这些静态方法打包成一个文件——StdArrayIO.java，就可以使我们轻松地复用代码，从而避免在稍后编写客户程序时再次编写读取和输出数组的细节。

237

程序2.2.2 数组I/O库

```

public class StdArrayIO
{
    public static double[] readDouble1D()
    {
        /*见练习2.2.11 */
    }

    public static double[][] readDouble2D()
    {
        int m = StdIn.readInt();
        int n = StdIn.readInt();
        double[][] a = new double[m][n];
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                a[i][j] = StdIn.readDouble();
        return a;
    }

    public static void print(double[] a)
    {
        /*见练习2.2.11 */
    }

    public static void print(double[][] a)
    {
        int m = a.length;
        int n = a[0].length;
        System.out.println(m + " " + n);
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
                StdOut.printf("%9.5f ", a[i][j]);
            StdOut.println();
        }
        StdOut.println();
    }

    // 针对其他类型的数据的方法是类似的(参见本书官网)
    public static void main(String[] args)
    {
        print(readDouble2D());
    }
}

```

```
% more tiny2D.txt
```

```
4 3
```

```
.000 .270 .000
```

```
.246 .224 -.036
```

```
.222 .176 .0893
```

```
-.032 .739 .270
```

```
% java StdArrayIO < tiny.txt
```

```
4 3
```

```
0.00000 0.27000 0.00000
```

```
0.24600 0.22400 -0.03600
```

```
0.22200 0.17600 0.08930
```

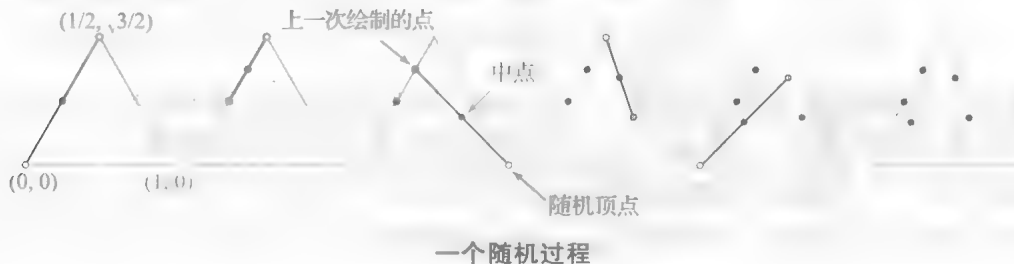
```
-0.03200 0.73900 0.27000
```

这种静态方法库有助于从标准输入读取一维和二维数组，并将它们打印到标准输出。文件格式包括维度（参见相应文本）。示例中输出的数字是截取的一部分。

238

迭代函数系统 科学家发现，有些复杂的视觉图像可以通过多次简单的计算过程得到。我们可以使用 `StdRandom`、`StdDraw` 和 `StdArrayIO` 研究这样的系统行为。

谢尔宾斯基三角形。我们首先来分析以下简单过程：一开始，在给定等边三角形的一个顶点上绘制一个点，然后在三个顶点中随机选择一个，并在刚绘制的点和前一个绘制的顶点的连线中点上绘制一个新点。不断重复这一操作。每次迭代中，我们会将上一次迭代时绘制出的线段中点作为起点，从三角形中随机选取一个顶点作为终点，以这两个点为线段两端，我们绘制出这条线段的中点。由于我们进行随机选择，所以这些点应该具有随机点的一些特征，在最初的几次迭代之后如下：

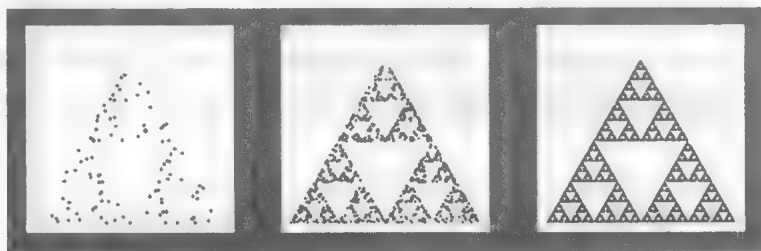


为了研究上述过程大量迭代后的结果，我们可以按照规则编写程序，并绘制出 `trails` 个点：

```
double[] cx = { 0.000, 1.000, 0.500 };
double[] cy = { 0.000, 0.000, 0.866 };
double x = 0.0, y = 0.0;
for (int t = 0; t < trials; t++)
{
    int r = StdRandom.uniform(3);
    x = (x + cx[r]) / 2.0;
    y = (y + cy[r]) / 2.0;
    StdDraw.point(x, y);
}
```

239

我们将三角形顶点的 x 坐标和 y 坐标分别存储在数组 `cx[]` 和 `cy[]` 中。我们通过 `StdRandom.uniform()` 从这些数组中随机选择一个下标索引 r ——所选顶点的坐标即为 $(cx[r], cy[r])$ 。从 (x, y) 到该顶点的线段的中点的 x 坐标由表达式 $(x+cx[r])/2.0$ 给出，同理可得 y 坐标。然后再通过对 `StdDraw.point()` 的调用将这个点绘制出来，将这些代码放到循环里，即可实现我们想要的功能。值得注意的是，尽管有随机性，但是经过大量的迭代，总是会出现相同的图像。这个图像被称为谢尔宾斯基三角形 (Sierpinski triangle) (见练习 2.3.27)。这样一个规则的图像是由这样的随机过程产生的，如何理解这个过程是一个非常有趣的问题。



一个随机过程？

巴恩斯利蕨。为了增加神秘性，我们可以改造这个游戏的规则，以创造出更丰富的图像。而最有趣的一个例子就是巴恩斯利蕨 (Barnsley fern)。想要生成这个图像，我们可以利用与前面相同的过程，但是这次需要使用下表的公式来完成计算。每一步，我们按照指定的概率来选择公式更新 x 和 y (使用第一对公式的概率是 1%，使用第二对公式的概率是 85%，以此类推)。

概率	更新 x 使用的公式	更新 y 使用的公式
1%	$x = \dots \quad 0.500$	$y = \dots \quad 0.16y$
85%	$x = 0.85x + 0.04y + 0.075$	$y = -0.04x + 0.85y + 0.180$
7%	$x = 0.20x - 0.26y - 0.400$	$y = 0.23x + 0.22y + 0.045$
7%	$x = -0.15x + 0.28y + 0.575$	$y = 0.26x + 0.24y - 0.086$

240

我们可以按照刚刚为谢尔宾斯基三角形写的代码那样的形式编写代码来迭代这些规则，但矩阵处理为我们提供了一种统一的方法以生成代码来处理任何规则集。如果有 m 种不同的转换，我们可以用一个 m 维向量表示每种转换的使用概率，然后我们用 `StdRandom.discrete()` 函数从中选择某一种转换方式。对于每个变换中 x 和 y 的更新方法，我们可以分

别为其定义方程，然后我们就可以使用两个 $m \times 3$ 矩阵来存储这些方程的系数，一个用于 x ，一个用于 y 。

程序2.2.3 迭代函数系统

```
public class IFS
{
    public static void main(String[] args)
    {
        // 进行trials遍绘制迭代，其中trials由标准输入得到
        int trials = Integer.parseInt(args[0]);
        double[] dist = StdArrayIO.readDouble1D();
        double[][] cx = StdArrayIO.readDouble2D();
        double[][] cy = StdArrayIO.readDouble2D();
        double x = 0.0, y = 0.0;
        for (int t = 0; t < trials; t++)
        {
            // 绘制一次迭代
            int r = StdRandom.discrete(dist);
            double x0 = cx[r][0]*x + cx[r][1]*y + cx[r][2];
            double y0 = cy[r][0]*x + cy[r][1]*y + cy[r][2];
            x = x0;
            y = y0;
            StdDraw.point(x, y);
        }
    }
}
```

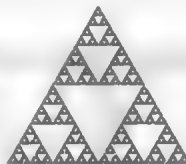
trials	迭代总数
dist[]	概率
cx[][]	x系数
cy[][]	y系数
x, y	当前点

这是一段数据驱动的程序，它是StdArrayIO、StdRandom和StdDraw的客户程序。程序从标准输入上迭代读取了一个 m 维向量（概率值）和两个 $m \times 3$ 矩阵（分别用于表示 x 和 y 的系数），然后基于这些参数所定义的函数系统进行迭代，最后在标准绘图上绘制一系列点。有趣的是，该代码不需要知道 m 的值，因为它使用另外的方法来创建和处理矩阵。

```
% more sierpinski.txt
```

```
3
.33 .33 .34
3 3
.50 .00 .00
.50 .00 .50
.50 .00 .25
3 3
.00 .50 .00
.00 .50 .00
.00 .50 .433
```

```
% java IFS 10000 < sierpinski.txt
```



```
% more barnsley.txt
```

```
4
.01 .85 .07 .07
4 3
.00 .00 .500
.85 .04 .075
.20 -.26 .400
-.15 .28 .575
4 3
.00 .16 .000
-.04 .85 .180
.23 .22 .045
.26 .24 -.086
```

```
% java IFS 20000 < barnsley.txt
```




迭代函数系统的一些例子

```

% more tree.txt
6
.1 .1 .2 .2 .2 .2
6 3
.00 .00 .550
-.05 .00 .525
.46 -.15 .270
.47 -.15 .265
.43 .26 .290
.42 .26 .290
6 3
.00 .60 .000
-.50 .00 .750
.39 .38 .105
.17 .42 .465
-.25 .45 .625
-.35 .31 .525

% java IFS 20000 < tree.txt

```

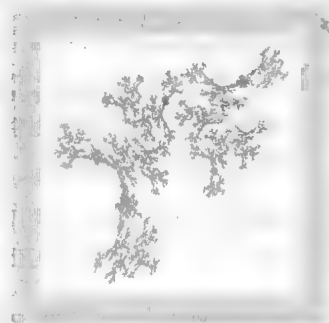


```

% more coral.txt
3
.40 .15 .45
3 3
.3077 -.5315 .8863
.3077 -.0769 .2166
.0000 .5455 .0106
3 3
-.4615 -.2937 1.0962
.1538 -.4476 .3384
.6923 -.1958 .3808

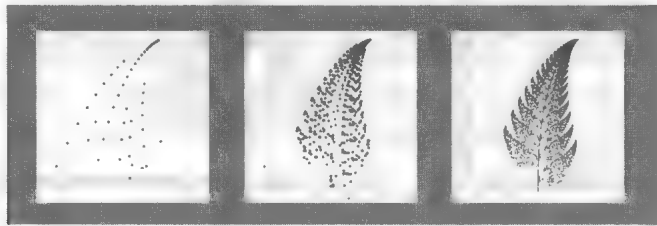
% java IFS 20000 < coral.txt

```



迭代函数系统的一些例子 (续)

IFS (程序 2.2.3) 实现了该计算的数据驱动版本。这个程序实现了无限的探索可能性：它能够根据任何有效输入数据进行迭代，输入数据包含一个定义概率分布的向量和两个定义系数的矩阵，分别用于描述 x 和 y 的更新方法。对于刚刚给出的方程系数，即使我们在每个步骤中选择随机方程，经过多次计算后都会出现相同的图像：这是一个看起来非常类似于你在树林里看到的蕨类植物，而不像是计算机上随机产生的事物。



生成一个巴恩斯利蕨

上述这样一个简短的程序 IFS，它从标准输入读取一些数字，然后根据不同的数据就可以在标准图像上绘制，不必修改任何代码就可以生成谢尔宾斯基三角形和巴恩斯利蕨（以及许多其他图像），这是非常引人注目的。因为代码简单而且结果神奇，这种计算对于图像合成等任务非常有用，常用于生成电影和游戏模拟现实图像。

也许更重要的是，通过如此简单的计算就能产生如此逼真的图像，为我们提出了一个有趣的科学问题：这个计算的过程是不是贴合某个自然规律？自然规律是不是遵从某个计算原理？

统计 接下来，我们讨论一个应用于科学与工程领域的库，它由一组并非全都可以在 Java 标准库里实现的数学计算和基本可视化工具组成。这些计算涉及理解一组数字的统计特性。这些库非常实用，例如，在我们执行一系列数据测量的科学实验时，就需要使用这些库

来处理数据。现代科学家面临的最重要的挑战之一就是对这些数据进行适当分析，而计算在这种分析中发挥着越来越重要的作用。下面的表格列出了这些基本数据分析方法的 API：

<code>public class StdStats</code>	
<code>double max(double[] a)</code>	最大值
<code>double min(double[] a)</code>	最小值
<code>double mean(double[] a)</code>	平均值
<code>double var(double[] a)</code>	样本方差
<code>double stddev(double[] a)</code>	样本标准差
<code>double median(double[] a)</code>	中位数
<code>void plotPoints(double[] a)</code>	在(i,a[i])处绘制点
<code>void plotLines(double[] a)</code>	绘制连接点(i,a[i])的线段
<code>void plotBars(double[] a)</code>	绘制(i,a[i])处的条形图

注：也包含其他数值类型的重载实现

数据分析的静态方法库的 API

程序2.2.4 数据分析库

```
public class StdStats
{
    public static double max(double[] a)
    { // 计算a[]的最大值
      double max = Double.NEGATIVE_INFINITY;
      for (int i = 0; i < a.length; i++)
        if (a[i] > max) max = a[i];
      return max;
    }

    public static double mean(double[] a)
    { // 计算a[]的平均值
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
        sum = sum + a[i];
      return sum / a.length;
    }

    public static double var(double[] a)
    { // 计算a[]值的样本方差
      double avg = mean(a);
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
        sum += (a[i] - avg) * (a[i] - avg);
      return sum / (a.length - 1);
    }

    public static double stddev(double[] a)
    { return Math.sqrt(var(a)); }

    // 参见程序2.2.5的绘制方法
    public static void main(String[] args)
    { /* 参见正文 */ }
}
```

该代码实现了计算客户端数组中数字的最大值、均值、方差和标准差的方法，而省略了计算最小值的方法，绘图方法在程序2.2.5中；median()的实现参见练习4.2.20。

```
% more tiny1D.txt
5
3.0 1.0 2.0 5.0 4.0
```

```
% java StdStats < tiny1D.txt
min    1.000
mean   3.000
max     5.000
std dev 1.581
```


基本统计。假设我们有 n 个测量值 x_0, x_1, \dots, x_{n-1} 。这些测量的平均值由公式 $\mu = \frac{(x_0 + x_1 + \dots + x_{n-1})}{n}$

给出，通常用来做这些数值的估计值。最小值和最大值也是有意义的，中位数（小于一半的数并且大于一半的数）的值也是如此。样本方差也是有用的，它由公式 $\sigma^2 = \frac{(x_0 - \mu)^2 + (x_1 - \mu)^2 + \dots + (x_{n-1} - \mu)^2}{(n-1)}$

计算；样本标准差是样本方差的平方根。StdStats（程序 2.2.4）显示了计算这些基本统计信息的静态方法（中位数比其他数据更难计算，我们将在 4.2 节中考虑 median() 的实现）。StdStats 的 main() 测试客户程序从标准输入读取数字到数组中，并调用打印最小值、平均值、最大值和标准偏差的方法，如下所示：

```
public static void main(String[] args)
{
    double[] a = StdArrayIO.readDouble1D();
    StdOut.printf("    min %7.3f\n", min(a));
    StdOut.printf("    mean %7.3f\n", mean(a));
    StdOut.printf("    max %7.3f\n", max(a));
    StdOut.printf("    std dev %7.3f\n", stddev(a));
}
```

与 StdRandom 一样，需要进行更广泛的计算测试才能证明程序的正确性（参见练习 2.2.3）。通常，当我们调试或测试库中的新方法时，我们相应地调整单元测试部分的代码，单元测试是指一次测试一种方法。一个像 StdStats 这样成熟和广泛使用的库也需要一个压力测试的客户程序，在每次代码发生修改后都需要对所有内容进行详细的测试。如果你有兴趣看看这样的客户程序是什么样的，你可以在本书官网上找到 StdStats 程序。大多数经验丰富的程序员认为花在单元测试和压力测试上的时间都是很值得的。

绘图。StdDraw 的一个重要用途是帮助我们实现数据的可视化，而不用依赖数据表格获取信息。在大多数情况下，我们执行实验，将实验数据保存在数组中，然后将结果与模型进行比较，或者与一个描述数据的数学函数进行比较。对于自变量的值是按照等差数列排列的情况，我们的 StdStats 库包含绘制数组中的数据的静态方法。程序 2.2.5 是 StdStats 的 plotPoints()、plotLines() 和 plotBars() 方法的实现。这些方法在绘图窗口中以均匀的间隔显示参数数组中的值，或者使用线段（line）连接在一起，或者使用每个值对应的圆点（point）填充，或者使用从 x 轴到该值的条柱（bar）。然后通过以上这些图形，使用 x 坐标 i 和 y 坐标 $a[i]$ 绘制这些点。此外，它们都重新缩放 x 以填充绘图窗口（使得点沿 x 坐标均匀间隔），然后把 y 坐标的缩放留给客户程序自己来随意控制。

这些方法的目标并不是提供一个通用的绘图软件包，但你可以很自然地找到可能需要添加的各种各样的工具：不同类型的点、坐标轴上的标签、颜色以及现代数据绘制系统的其他工具。在有些情况下甚至可能需要比这些更复杂的方法。

我们使用 StdStats 的意图是向你介绍数据分析的相关知识，同时向你介绍如何轻松地定义库来完成数据分析任务。实际上，这个库已经证明是有用的——我们就是使用这些绘图方法生成了本书后面几节的插图，分别是：绘制函数图像、绘制声波和绘制实验结果。接下来，我们来讨论这几个使用示例。

绘制函数图。你可以使用 StdStats.plot*() 方法来绘制任何函数的图像：选择要在其中绘制函数的 x 区间，在该区间内，按照均匀的间隔计算对应的函数值，并把它们存储在数组中，确定并设置 y 轴的缩放系数，然后调用 StdStats.plotLines() 或者其他 plot*() 方法。例如，要绘制正弦函数，需要设置 y 轴的范围为 -1 到 $+1$ 之间。 x 轴的范围由 StdStats 方法自动设定。如果你不了解该范围，可以调用以下方法：

```
StdDraw.setXscale(StdStats.min(a), StdStats.max(a));
```

程序2.2.5 绘制数组中的数据值

```

public static void plotPoints(double[] a)
{ // 在(i,a[i])处绘制点
  int n = a.length;
  StdDraw.setXscale(-1, n);
  StdDraw.setPenRadius(1/(3.0*n));
  for (int i = 0; i < n; i++)
    StdDraw.point(i, a[i]);
}

public static void plotLines(double[] a)
{ // 绘制通过点(i,a[i])的线
  int n = a.length;
  StdDraw.setXscale(-1, n);
  StdDraw.setPenRadius();
  for (int i = 1; i < n; i++)
    StdDraw.line(i-1, a[i-1], i, a[i]);
}

public static void plotBars(double[] a)
{ // 绘制一个连接(0,a[i])和 (i,a[i])的条柱
  int n = a.length;
  StdDraw.setXscale(-1, n);
  for (int i = 0; i < n; i++)
    StdDraw.filledRectangle(i, a[i]/2, 0.25, a[i]/2);
}

```

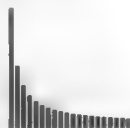
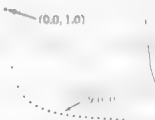
此代码在StdStats（程序2.2.4）中实现了绘制数据的三种方法。它们分别用实心圆、线段和条柱来表示点(i,a[i])。

```

int n = 20;
double[] a = new double[n];
for (int i = 0; i < n; i++)
  a[i] = 1.0/(i+1);

```

plotPoints(a); plotLines(a); plotBars(a);



曲线的平滑度由函数的性质和绘制的点数确定。正如我们在首次讲到 StdDraw 时所讨论的，你必须小心地抽取足够的点以体现函数的波动。稍后在 2.4 节中，我们还将用另一种方法来绘制不等间距采样值的函数。

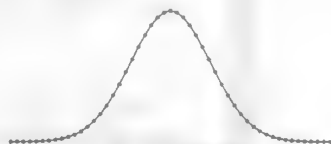
绘制声波。 StdAudio 库和 StdStats 绘图方法处理包含规律间隔的采样值的数组。1.5 节和本节开始时的声波图形都是首先用 StdDraw.setYscale(-1,1) 缩放 y 轴，然后用 StdStats.plotPoints() 绘制点。正如你所看到的，这样的图像可以直观显示音频处理的效果。你也可以在使用 StdAudio 播放声波的同时把声波绘制出来。这样能够产生有趣的效果，不过由于涉及的数据量很大，这项任务有一点挑战性（参见练习 1.5.23）。

绘制实验结果。 你可以在同一图纸上放置多个绘图，一般这样做的目的是为了将实验结果与理论模型进行比较。例如，Bernoulli（程序 2.2.6）计算一枚普通硬币翻

```

int n = 50;
double[] a = new double[n+1];
for (int i = 0; i <= n; i++)
  a[i] = Gaussian.pdf(-4.0 + 8.0*i/n);
StdStats.plotPoints(a);
StdStats.plotLines(a);

```

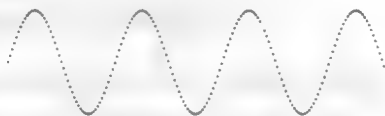


绘制函数图

```

StdDraw.setYscale(-1.0, 1.0);
double[] hi;
hi = PlayThatTune.tone(880, 0.01);
StdStats.plotPoints(hi);

```



绘制声波

转 n 次正面朝上的次数，并将结果与预测的高斯概率密度函数进行比较。概率论的一个著名结论是：虽然这个实验结果是二项分布（binomial distribution），但它非常近似于均值为 $n/2$ 、标准偏差为 $\frac{\sqrt{n}}{2}$ 的高斯分布。我们执行的试验次数越多，两个分布的偏差越小。Bernoulli 生成的图像是对实验结果的简单总结，并对理论进行了令人信服的验证。这个例子是科学方法转化成为应用程序编程的原型，我们在本书中经常使用。建议你每次运行实验时也应该使用它。如果一个理论模型可以用来解释实验结果，则可以通过数据的可视化将实验结果与理论模型进行比较，以实现二者的相互验证。

这几个例子旨在说明，借助精心设计的静态方法库可以对数据分析产生怎样的帮助，练习中还展示了其他几个扩展和想法。你将发现，StdStats 对于基本图形的绘制非常有用，我们鼓励你尝试实验它们，然后修改或者添加一些方法，以创建自己的库来绘制设计图。当你继续解决越来越广泛的编程任务时，你将自然而然地开始开发满足自己需求的编程工具库。

249

程序2.2.6 伯努利试验

```
public class Bernoulli
{
    public static int binomial(int n)
    { // 模拟翻转硬币n次; 返回正面次数
        int heads = 0;
        for (int i = 0; i < n; i++)
            if (StdRandom.bernoulli(0.5)) heads++;
        return heads;
    }
    public static void main(String[] args)
    { // 执行伯努利试验, 绘制结果和模型
        int n = Integer.parseInt(args[0]);
        int trials = Integer.parseInt(args[1]);

        int[] freq = new int[n+1];
        for (int t = 0; t < trials; t++)
            freq[binomial(n)]++;

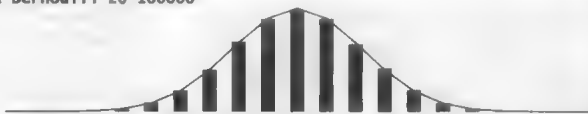
        double[] norm = new double[n+1];
        for (int i = 0; i <= n; i++)
            norm[i] = (double) freq[i] / trials;
        StdStats.plotBars(norm);

        double mean = n / 2.0;
        double stddev = Math.sqrt(n) / 2.0;
        double[] phi = new double[n+1];
        for (int i = 0; i <= n; i++)
            phi[i] = Gaussian.pdf(i, mean, stddev);
        StdStats.plotLines(phi);
    }
}
```

n	每次试验的翻转次数
trials	试验次数
freq[]	实验结果
norm[]	标准化结果
phi[]	高斯模型

StdStats, StdRandom和Gaussian客户端可视化说明了：翻转硬币 n 次，其正面朝上的次数符合高斯分布。

% java Bernoulli 20 100000



250

模块化编程 我们开发的库展现了一种称为模块化编程（modular programming）的编程风格。相比于在一个单独的程序文件中编写一个包含所有功能细节的新程序来解决新问题，

我们更加倾向于将每个任务分解成更小、更易于管理的子任务，然后实现和独立调试解决每个子任务的代码。好的库允许我们为未来的客户定义和提供重要子任务的解决方案，以促进模块化编程。在编程时，要尽可能地将程序明确地分割成相互独立的子任务，然后分别实现。Java 支持这种分离的开发模式，具体表现为 Java 允许一个文件中一个类进行独立开发和调试，稍后可以将多个类对应的文件集中在一起使用。通常，程序员使用术语模块（module）来指代那些可以独立编译和运行的代码；在 Java 中，每个类（class）都是一个模块。

IFS（程序 2.2.3）就是一个模块化编程的例子。这种相对复杂的计算是通过独立开发的几个相对较小的模块实现的。它使用 StdRandom 和 StdArrayIO，以及我们习惯使用的 Integer 和 StdDraw 中的方法。如果我们将 IFS 所需的所有代码放在一个文件中，我们将有大量的代码要维护和调试；通过模块化编程，我们就可以更有信心研究迭代函数系统，因为我们已经在单独的模块中实现并测试了这些任务的代码，因此可以确信这些代码可以正确读取数组，并且随机数生成器也能产生正确分布的值。

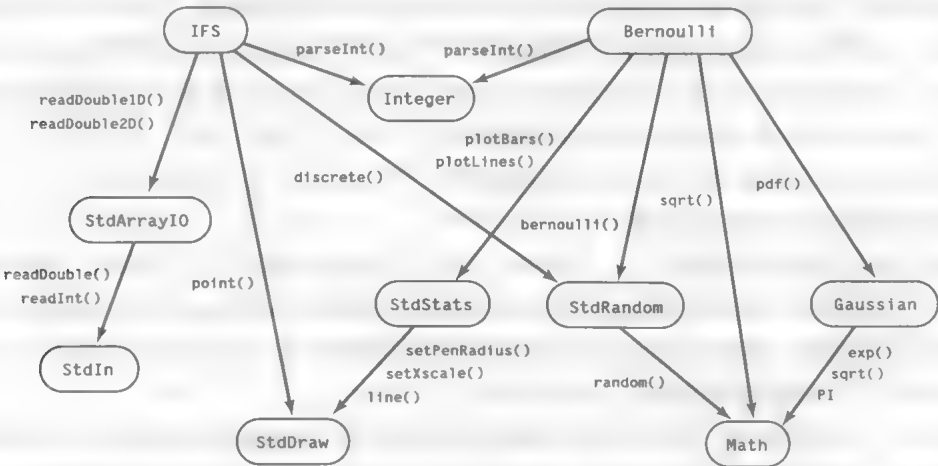
API	描述
Gaussian	高斯分布函数
StdRandom	随机数
StdArrayIO	数组的输入输出
IFS	迭代函数系统的客户程序
StdStats	数据分析函数
Bernoulli	伯努利试验的客户程序

本节中类的汇总

类似地，Bernoulli（程序 2.2.6）也是一个模块化编程的例子。它是 Gaussian、Integer、Math、StdRandom 和 StdStats 的客户程序。我们可以再次对这些模块中的方法产生的预期结果有一些信心，因为它们是系统库或我们以前测试、调试和使用过的库。

251

为了描述模块化程序中模块之间的关系，我们通常会绘制一个依赖关系图（dependency graph），如果第一个类包含一个方法的调用，而第二个类包含该方法的定义，那么我们用这个方法命名的箭头连接这两个类名。这样的图表对于代码的开发和维护起着重要的作用，因为理解模块之间的关系是十分必要的。



本节模块的关系图（部分）

我们在本书中强调模块化编程，因为它具有许多被认为是现代编程必不可少的重要优点，包括：

- 即使在大型系统中，我们也应该合理地控制程序的大小。
- 调试小块代码更加容易。
- 我们可以复用代码，而无须重新实现。
- 维护（和改进）代码要简单得多。

我们将展开来讨论。

合理控制程序的大小。没有任何大的任务是复杂到不能被分割成更小的子任务的。如果你发现自己的程序已经大到需要多页代码才能完成，你必须问自己如下问题：这些子任务是否可以单独实现？这些子任务中的某部分是否可以逻辑分组到一个单独的库中？其他客户程序将来是否可以使用此代码？反过来说，如果你发现自己的模块太多、太小，你也必须问自己如下问题：是否有一些逻辑上属于同一个模块的子任务？每个模块是否可能被多个客户程序使用？对于模块大小来说，没有必须遵守的规则：一个重要的抽象概念的实现可能恰恰是几行代码，而另一个具有大量重载方法的库可能会适当地扩展到数百行代码。

[252]

调试。随着语句和交互变量的增多，追踪程序变得更加困难。追踪具有数百个变量的程序需要追踪数百个值，因为任何语句都可能会影响其中一个变量的值，或受到某个变量的影响。追踪成百上千甚至更多的语句是无法实施的。模块化编程和将变量控制在可控范围内的指导原则，促使我们在必须调试时可以严格限制出错的范围。客户程序和实现之间的契约同样重要。一旦我们确信一段代码实现了所有任务，那么我们就可以把这个条件推广到任何客户程序的调试中（即在调试客户程序时确信库程序没有错误——译者注）。

代码复用。一旦我们实现了 `StdStats` 和 `StdRandom` 这样的库，我们就不必编写代码来计算平均值或标准差，或者生成随机数，而可以只是简单地复用这些代码。此外，我们也不需要复制代码：所有模块都可以引用其他任一模块中的所有公共方法。

维护。如同写作一样，好的程序总是可以被不断修改，而模块化编程便于不断改进 Java 程序，因为改进一个模块就可以改进所有的客户程序。例如，在通常情况下，解决特定的问题有几种不同的方法。使用模块化编程，你可以实现多个独立实现。更重要的是，在开发新客户程序的同时，你可能会发现某些模块中的错误。因为模块化编程，修复该模块中的错误基本上修复了所有客户程序中的同类错误。

[253]

如果你遇到一个旧程序（或者一个老套的程序员编写的新程序），你可能会发现一个巨大的模块——长串的语句，扩展到几页或更多页，其中任何语句可以引用该程序的任何变量。这些老式程序甚至会出现我们的基础设施的计算程序的关键部分（如某些核电厂和部分银行），这时因为负责维护的程序员无法理解这种写法，导致了他们无法用现代语言对其进行重写。通过支持模块化编程，Java 这样的现代语言可通过将功能划分成类，然后单独开发方法库来帮助我们避免这种情况。

在不同文件之间共享静态方法，从根本上扩展了我们的编程模型。第一，它允许我们复用代码，而不必维护它的多个副本；第二，它允许我们将程序组织成可以独立编程、易于调试、大小合理的文件。这两点都强烈地支持了我们的基本理念：在编程时，要尽可能地将程序明确地分割成相互独立的子任务，然后分别实现。

在本节中，我们在 1.5 节的几个库的基础上又增加了 `Gaussian`、`StdArrayIO`、`StdRandom` 和 `StdStats` 等一些可供使用的库。同时，我们用几个客户程序说明了它们的用法。这些工具

主要是在科学项目或工程任务中出现的基本数学概念。本书的目的不仅仅是提供工具，还要说明创建自己的工具也是很容易的。大多数现代程序员在处理复杂任务时提出的第一个问题是“我需要哪些工具？”。当需要的工具不方便得到时，第二个问题是“实现它们有多难？”。要成为一个好的程序员，你应该有信心构建自己的软件工具，并且也应该足够聪明地知道何时该去库中寻求更好的解决方案。

在学习了库和模块化编程后，学习现代编程模型还有一个步骤：面向对象的编程（object-oriented programming），也就是第3章的主题。通过面向对象编程，你可以构建充分发挥函数的副作用的库（当然，是以一种受控的方式来使用的），以极大扩展 Java 编程模型。在转向面向对象编程之前，在本章中我们还将涉及以下两方面的内容：任何方法都可以调用自身的编程理念（2.3 节），以及在较大规模的客户程序中关于模块化编程的案例研究（2.4 节）。 254

问答环节

问：我试图使用 StdRandom，但是收到了错误消息“Exception in thread” main “java.lang.NoClassDefFoundError: StdRandom”，错在哪里了呢？

答：你需要使 Java 可以访问 StdRandom。请看 1.5 节末尾问答环节的第一问。

问：是否有一个关键字能够将 class 标识为库？

答：没有，因为类中的任何一个 public 类型的方法都可以被定义为库中的函数。从这个角度来说，有一点概念上的脱节。确实，创建一个你自己编译和运行的 .java 文件是一回事，创建一个自己很久以后才会使用的 .java 文件是另一回事，而创建一个 .java 文件供别人在将来使用又是完全不同的另一回事了。你需要多写一些库来供自己使用，然后才能成长为有经验的系统程序员。

问：如何开发一个我已经使用了一段时间的库的新版本？

答：谨慎。我们最好在单独的目录中工作，因为对 API 的任何更改都可能会破坏原来的客户程序。所以当你修改你的库时，确保你正在处理的是代码的副本。当你改变一个有许多客户的库时，你就能明白许多公司在推出新版本的软件时面临的问题了。如果你只是想给库添加一些方法，那就直接做吧：这样做不太危险，但你经过几次修改后就会意识到，你可能还需要支持这个库很多年！

问：我要怎么知道一个实现是否正确地运行了？为什么不自动检查它是否满足 API？

答：我们使用非正式的说明，因为要编写详细的说明和直接编写程序没什么区别。此外，理论计算机科学的基本原理表明，这样做甚至无法解决基本问题，因为通常没有办法检查两个不同的程序是否执行了相同的计算。 255

练习

- 2.2.1 在 Gaussian（程序 2.1.2）中增加一个三参数静态方法 pdf(x,mu,sigma) 的实现，使其根据给定的平均值 μ 和标准差 σ ，根据公式 $\phi(x,\mu,\sigma)=\phi((x-\mu)/\sigma)/\sigma$ ，计算高斯概率密度函数。再基于公式 $\Phi(z,\mu,\sigma)=\Phi((z-\mu)/\sigma)$ ，增加一个相关的累积分布函数 cdf(z,mu,sigma) 的实现。
- 2.2.2 编写一个静态方法库以实现双曲线函数，其中：定义 $\sinh(x)=(e^x-e^{-x})/2$ ， $\cosh(x)=(e^x+e^{-x})/2$ 的，其中 $\tanh(x)$ 、 $\coth(x)$ 、 $\operatorname{sech}(x)$ 和 $\operatorname{csch}(x)$ 以类似于标准三角函数的方式定义。
- 2.2.3 为 StdStats 和 StdRandom 编写一个测试用客户程序，以检查两个库中的方法是否按预期运行。该程序需要输入命令行参数 n，使用 StdRandom 中的每种方法生成 n 个随机数，并打印其统计

信息。提示：将计算得到的结果与预期的结果进行比较以得出结论。

- 2.2.4 为 `StdRandom` 添加一个方法 `shuffle()`，它将一个 `double` 型数组作为参数，并以随机顺序不断重新排列。设计一个测试用客户程序，检查数组的每个排列出现的次数是否大致相同。将参数替换为 `Int` 型数组和 `Strings` 型数组并给出重载后的设计。
- 2.2.5 开发一个对 `StdRandom` 进行压力测试的客户程序。要特别注意 `discrete()`。例如，概率总和是否为 1？
- 2.2.6 写一个静态方法，参数是 `double` 型的 `ymin` 和 `ymax`（`ymin` 严格小于 `ymax`）和一个 `double` 数组 `a[]`，并使用 `StdStats` 库线性缩放 `a[]` 中的值，使 `a[]` 中每个元素都在 `ymin` 和 `ymax` 之间。
- 2.2.7 编写 `Gaussian` 和 `StdStats` 的客户程序，探讨改变高斯概率密度函数的均值和标准差的影响。在均值固定、标准差不同的情况下，画高斯分布的图像；在标准差固定、均值不同的情况下，画高斯分布的图像。
- 2.2.8 为 `StdRandom` 添加方法 `exp()`，参数为 λ ，返回一个服从参数为 λ 的指数分布的随机数。提示：如果 x 是在 0 和 1 之间均匀分布的随机数，则 $-\ln x / \lambda$ 就是参数为 λ 的指数分布返回的随机数。
- 2.2.9 向 `StdRandom` 添加一个静态方法 `maxwellBoltzmann()`，返回一个服从参数为 σ 的 Maxwell-Boltzmann 分布的随机值。为了产生这样的值，从均值为 0、标准差为 σ 的高斯分布得到三个随机数，取其平方和的平方根。理想气体中分子的速度符合 Maxwell-Boltzmann 分布。
- 2.2.10 修改 `Bernoulli`（程序 2.2.6），以动画形式生成条形图。在每次实验后重新绘制，以便你可以观察它收敛于高斯分布。然后添加命令行参数，用于设定硬币正面朝上的概率 p ，并重载 `binomial()` 的实现。一定要试试 p 的值趋近 0 以及趋近 1 的情况。
- 2.2.11 开发 `StdArrayIO` 的所有实现（实现 API 中指出的所有 12 种方法）。
- 2.2.12 编写一个实现以下 API 的库 `Matrix`：

```
public class Matrix
```

<code>double dot(double[] a, double[] b)</code>	矢量点积
<code>double[][] multiply(double[][] a, double[][] b)</code>	矩阵-矩阵积
<code>double[][] transpose(double[][] a)</code>	转置
<code>double[] multiply(double[][] a, double[] x)</code>	矩阵-向量积
<code>double[] multiply(double[] x, double[][] a)</code>	向量-矩阵积

（见 1.4 节）。作为测试用客户程序，使用以下代码执行与 `Markov`（程序 1.6.3）相同的计算：

```
public static void main(String[] args)
{
    int trials = Integer.parseInt(args[0]);
    double[][] p = StdArrayIO.readDouble2D();
    double[] ranks = new double[p.length];
    rank[0] = 1.0;
    for (int t = 0; t < trials; t++)
        ranks = Matrix.multiply(ranks, p);
    StdArrayIO.print(ranks);
}
```

数学家和科学家使用成熟的库或特殊用途的矩阵处理语言完成这些任务。有关使用此类库的详细信息请参阅本书官网。

- 2.2.13 编写实现 1.6 节中描述的 `Markov` 版本的 `Matrix` 客户程序，但是基于矩阵平方的迭代，而不是向量-矩阵相乘的迭代。

2.2.14 使用 StdArrayIO 和 StdRandom 库重写 RandomSurfer (程序 1.6.2)。

部分代码：

```
...
double[][] p = StdArrayIO.readDouble2D();
int page = 0; // 从第0页开始
int[] freq = new int[n];
for (int t = 0; t < trials; t++)
{
    page = StdRandom.discrete(p[page]);
    freq[page]++;
}
...
```

258

创新练习

2.2.15 Sicherman 骰子。假设你有两个六面体骰子，一个是标有 1、3、4、5、6、8，另一个被标记为 1、2、2、3、3、4。将骰子总和的每个值的发生概率与标准骰子对的概率进行比较。使用 StdRandom 和 StdStats。

2.2.16 骰子。滚动两个六面骰子， x 为它们的总和。以下是掷骰子游戏中赌赢的规则。

- 如果 x 是 7 或 11，你赢了。
- 如果 x 为 2、3 或 12，你输了。

否则，重复滚动两个骰子，直到它们的总和为 x 或 7。

- 如果它们的总和为 x ，你赢了。
- 如果它们的总和是 7，你输了。

编写一个模块化程序来估计赢得赌注的可能性。然后，修改你的程序来处理以下情况：假设你可以控制骰子，从命令行获取骰子获得点数为 1 的概率，点数为 6 的概率为 $1/6$ 减去该概率，点数 2~5 被假定为同等可能，再次计算不同参数下你的获胜概论。提示：使用 StdRandom.discrete()。

2.2.17 高斯随机数。使用 Box-Muller 公式（见练习 1.2.27）在 StdRandom（程序 2.2.1）中实现无参数 gaussian() 函数。接下来，考虑一种称为 Marsaglia 方法的替代方法，该方法的思路基于在单位圆中生成随机点并使用 Box-Muller 公式的形式进行计算（参见 1.3 节结尾处的 do-while 的讨论）。

```
public static double gaussian()
{
    double r, x, y;
    do
    {
        x = uniform(-1.0, 1.0);
        y = uniform(-1.0, 1.0);
        r = x*x + y*y;
    } while (r >= 1 || r == 0);
    return x * Math.sqrt(-2 * Math.log(r) / r);
}
```

对于每种方法，从高斯分布生成 1000 万个随机值，并测量哪一个更快。

259

2.2.18 动态直方图。假设标准输入流是 double 序列。编写一个程序，它从命令行中读取一个整数 n 和两个 double 型值 lo 和 hi。把 (lo, hi) 分割为 n 个相等大小的间隔，用 StdStats 画一个直方图来描述标准输入流落在每个间隔中的个数。使用程序将代码添加到练习 2.2.3 的解决方案中，从命令行读入 n ，并绘制每个方法生成的数字分布的直方图。

- 2.2.19 压力测试。开发一个对 `StdStats` 进行压力测试的客户程序。与同学合作，一个人编写代码，另一个测试它。
- 2.2.20 赌徒破产问题。开发一个 `StdRandom` 客户程序来研究赌徒破产问题（见程序 1.3.8 和练习 1.3.24~1.3.25）。注意：为这个实验定义一个静态方法要比为 `Bernoulli` 定义难，因为你不能返回两个值。
- 2.2.21 IFS。实验 IFS 的各种输入，以创建你自己的设计模式，如谢尔宾斯基三角形、巴恩斯利蕨类植物或文本中的其他示例。你可以从修改给定输入开始。
- 2.2.22 IFS 矩阵实现。编写使用 `Matrix` 的静态方法 `multiply()` 的 IFS 版本（参见练习 2.2.12），而不是计算 `x0` 和 `y0` 的新值的方程式。
- 2.2.23 整数属性库。根据我们在本书中考虑的用于计算整数属性的函数来开发一个库。包括：确定给定整数是否为质数的函数；确定两个整数是否互质；计算给定整数的所有因子；计算最大公约数和两个整数的最小公倍数；欧拉函数（练习 2.1.26）；以及你认为可能有用的任何其他函数。并通过重载给出 `long` 型数的实现。创建一个 API、执行压力测试的客户程序，以及解决本书前面几个练习问题的客户程序。
- 2.2.24 音乐库。根据 `PlayThatTune`（程序 2.1.4）中的函数开发一个库，你可以使用它来编写客户程序以创建和操作歌曲。
- 2.2.25 投票机。开发一个 `StdRandom` 客户程序（具有它自己的必要的静态方法）来研究以下问题：假设在一亿个投票人中，51% 的人投给候选人 A，49% 的人投给候选人 B。但投票机容易出错，5% 的概率会产生错误的统计结果。假设这些错误是独立的、随机的，5% 的错误率是否会使得结果无效？可允许的最大错误率是多少？
- 2.2.26 扑克分析。写一个 `StdRandom` 和 `StdStats` 客户程序（具有它们自己的适当的静态方法），通过模拟计算一手牌（五张）中，获得一对、两对、三条、葫芦、同花的概率（你可能需要自行搜索这些牌的具体含义——译者注）。将你的程序划分为适当的静态方法，并论述你的设计决策的正确性。加分题：在概率列表中添加顺子和同花顺的概率。
- 2.2.27 动画图。编写一个程序，该程序需要输入命令行参数 `m`，并生成标准输入上最新的 `m` 个 `double` 型数的条形图。使用与 `BouncingBall`（程序 1.5.6）相同的动画技术：擦除、重绘、显示和短暂等待。每次程序读取一个新的数字时，都应该重绘整个条形图。由于图片的大部分内容没有改变，只是略微向左移动，你的程序会产生一个固定大小的窗口根据输入值动态滑过的效果。使用你的程序绘制大规模的根据时间变化的数据文件，比如股票价格。
- 2.2.28 数组绘制库。开发自己的绘图方法来改进 `StdStats` 中的方法。要有创意！尝试制作一个你认为将来对某些应用程序有用的绘图库。

2.3 递归

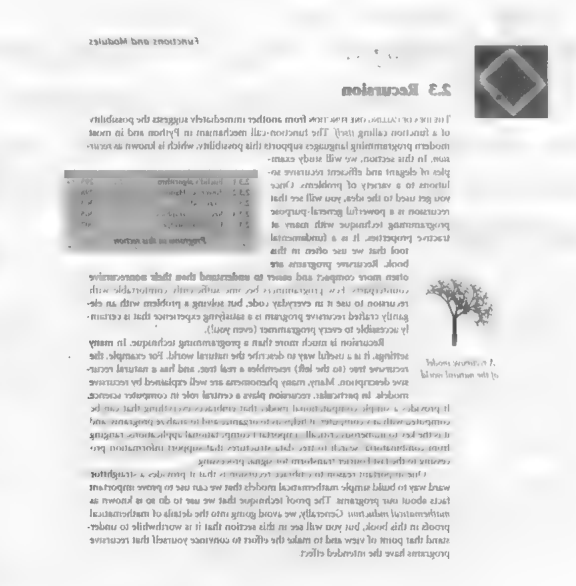
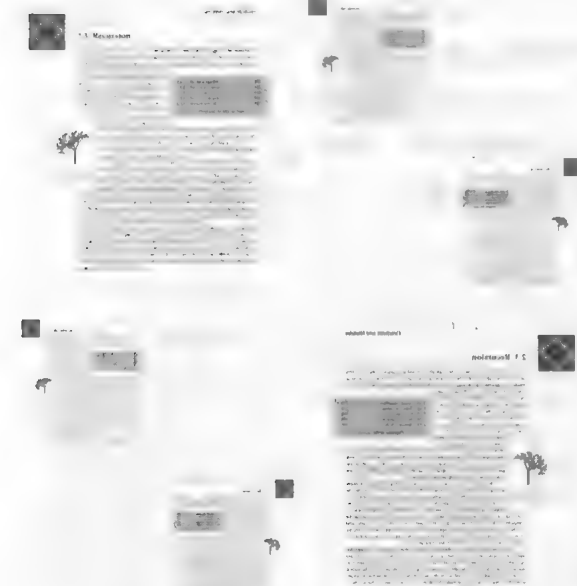
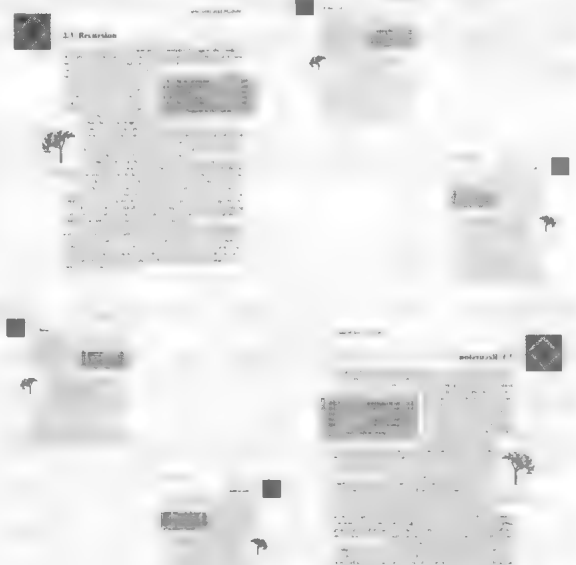
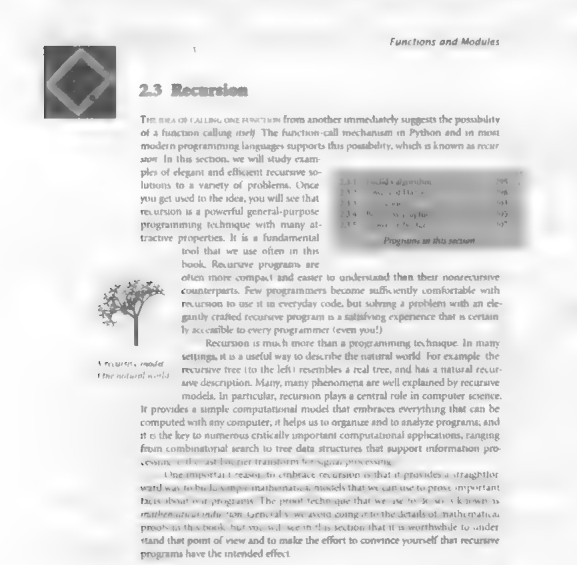
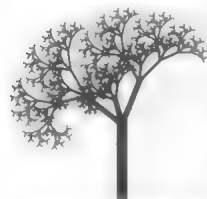
函数可以调用其他函数，这使我们立即想到函数还可以调用自身。Java 和大多数现代编程语言一样，在函数调用机制中支持函数调用自身。函数调用自身被称为递归（*recursion*）。在本节中，我们将研究各种问题的高效的递归解决方案。递归是在本书中经常使用的强大的编程技术。递归的程序通常比非递归程序更紧凑、更易于理解。很少有程序员能在短时间内非常熟练地使用递归，但用递归解决问题会是令人满足的体验，每个程序员都可以有这样的体验（你也可以！）。

递归不仅仅是编程技术。在许多环境中，它是描述真实世界的有效方法。例如，递归树（下图）类似于一棵真实的树，并且具有自然的递归特征。递归模型很好地解释了许多现象。

递归在计算机科学中更是起着核心作用，它提供了一个简单的计算模型，能够用于描述任何可以用计算机进行计算的内容；它帮助我们组织和分析程序；它是许多重要的计算应用的关键。递归应用的范围很广，无论是支持信息处理的树数据结构 的组合搜索，还是用于信号处理的快速傅里叶变换，都可以使用递归技术来实现。

支持递归技术的一个重要原因是它提供了一种简单直接的方法来构建自然界的递归模型 数学归纳 (mathematical induction) 的模型。数学归纳法是一种重要的数学证明手段，我们可以用它来证明很多重要的定理。在本书中，我们不再详细介绍数学证明，而只强调编程的方法，但是你们有必要理解这一过程，并明白递归程序在这一过程中的执行过程。

262



递归图像展示

263

你的第一个递归程序 如同第一个程序通常是“Hello,World”，第一个递归程序通常是计算阶乘（factorial）函数。正整数 n 的阶乘函数定义如下：

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

换句话说， $n!$ 是小于或等于 n 的正整数的乘积。现在，使用 for 循环计算 $n!$ 很容易，但更简单的方法是使用以下递归函数：

```
public static long factorial(int n)
{
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

此函数调用了自身，属于一个递归程序。你要相信它确实可以实现我们想要的功能。让我们来分析这个过程，当 n 为 1 时，factorial() 函数返回 1，这是个正确答案；如果它需要计算下式时

$$(n-1)! = (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

那么它可以通过下面的式子正确地计算出这个值

$$n! = n \times (n-1)! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

为了计算 factorial(5)，递归函数将 5 与 factorial(4) 相乘；计算 factorial(4)，将 4 与 factorial(3) 相乘；以此类推。这个过程一直重复，直到调用参数为 1 时，factorial(1) 直接返回 1。我们追溯这一系列的函数调用即追踪了整个计算过程。如果我们将所有被调用的函数视为相互独立的代码副本，那么，它们是否是递归也就没有本质区别。

factorial() 函数的实现展示了每个递归函数所需的两个主要部分。首先，函数的基础步骤是在没有任何递归调用的情况下返回一个值。基础步骤可以是对一个或多个特殊的输入值的操作，可

```
factorial(5)
  factorial(4)
    factorial(3)
      factorial(2)
        factorial(1)
          return 1
        return 2*1 = 2
      return 3*2 = 6
    return 4*6 = 24
  return 5*24 = 120
```

factorial(5) 的函数调用追踪

[264] 以在不递归的情况下对该函数进行求值。对于 factorial() 函数，基础步骤为 $n=1$ 。其次是归约步骤（reduction step），这是递归函数的核心部分。它将函数的一个（或多个）参数的调用转换成该函数的另一个（或多个）参数的调用，以及其他相关联的计算。对于 factorial() 函数，归约步骤是通过 $n * \text{factorial}(n-1)$ 实现的。所有递归函数必须具有这两个部分。此外，参数值序列必须收敛到基础步骤。对于 factorial() 函数，每个调用之后 n 减少 1，因此参数值序列收敛到最后是 $n=1$ 。

如果我们将归约步骤放在 else 子句中，那么像 factorial() 这样的小程序可能会变得更加清晰。然而，在递归程序中进行这样的修改并不一定是个好主意。因为对于稍微复杂的程序，绝大部分代码都是归约步骤，按照这样修改后，这些代码将放在 else 后的大括号内执行，只会使程序变得复杂而毫无收益。我们建议采用以基础步骤作为第一个语句，以返回作为结尾，然后将其余代码用于归约步骤，这样的表示更加清晰简练。

factorial() 的实现本身在实践中不是特别有用，因为 $n!$ 增长过快，乘法将溢出，并且在 $n > 20$ 时会产生不正确的答案。但同样的

```
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
11 39916800
12 479001600
13 6227020800
14 87178291200
15 1307674368000
16 20922789888000
17 355687428096000
18 6402373705728000
19 121645100408832000
20 2432902008176640000
n! 的求解过程
```

技术能有效地计算各类函数。例如递归函数

```
public static double harmonic(int n)
{
    if (n == 1) return 1.0;
    return harmonic(n-1) + 1.0/n;
}
```

这是一个用于当 n 较小时计算谐波数的程序（见程序 1.3.5），其计算公式如下：

$$\begin{aligned} H_n &= 1 + 1/2 + \cdots + 1/n \\ &= (1 + 1/2 + \cdots + 1/(n-1)) + 1/n \\ &= H_{n-1} + 1/n \end{aligned}$$

实际上，对于任何可以写出紧凑公式的任何离散求和（或求乘积）的运算，都可以用同样的方法计算，而且只需要几行代码。像这样的递归函数看起来像循环结构，但递归确实可以帮助我们更好地了解计算的过程。

265

数学归纳法 递归编程与数学归纳法直接相关，数学归纳法是一种广泛用于自然数的技术。

通过数学归纳法证明一个包含有整数 n 的命题为真（假设 n 有无限多个值），包括以下两个步骤：

- 基础步骤：证明 n 为某些特定值（通常为 0 或 1）时，命题为真。
- 归纳步骤（证明的中心部分）：假定对于所有小于 n 的正整数命题为真，据此推断出整个命题为真。

这样足以证明，对于 n 的无穷多值，该命题为真：我们可以从基础步骤开始，对于每一个基础步骤的 n ，逐一证明命题为真。

对于数学归纳法的学习我们通常从以下命题开始：所有小于或等于 n 的正整数的和是 $n(n+1)/2$ 。即：我们需要证明当 $n \geq 1$ 时，以下公式成立：

$$1 + 2 + 3 + \cdots + (n-1) + n = n(n+1)/2$$

因为 $1 = 1(1+1)/2$ ，所以对于 $n=1$ （基础步骤），这个公式是成立的。如果我们假设对于小于 n 的所有正整数都是成立的，即对于 $n-1$ 的情况也是成立的，所以

$$1 + 2 + 3 + \cdots + (n-1) = (n-1)n/2$$

我们可以将 n 加到这个公式的两边，并化简就得到所需公式（归纳步骤）。

每次编写一个递归程序时，我们需要通过数学归纳法来确信该程序具有预期的效果。归纳和递归之间的对应关系是不言而喻的。而命名的差异表明了其侧重点的不同：在递归程序中，我们的目的是通过归约到较小问题来完成计算，因此我们使用术语归约步骤；在归纳证明中，我们的目的是为了确定较大问题的命题的真实性，所以我们使用术语归纳步骤。

当编写递归程序时，我们通常不会写出一个完整的正式的证明过程来证明它们产生了所需的结果，但是我们总是假设已经做过了一个类似这样的证明。事实上，我们经常需要一个非正式的归纳证明使自己相信，递归程序能够按预期工作。例如，我们刚刚讨论了一个非正式的证明，以确信 `factorial()` 可计算小于或等于 n 的正整数的乘积。

266

程序2.3.1 欧几里得算法

```

public class Euclid
{
    public static int gcd(int p, int q)
    {
        if (q == 0) return p;
        return gcd(q, p % q);
    }
    public static void main(String[] args)
    {
        int p = Integer.parseInt(args[0]);
        int q = Integer.parseInt(args[1]);
        int divisor = gcd(p, q);
        StdOut.println(divisor);
    }
}

```

p, q	参数
divisor	最大公约数

```

% java Euclid 1440 408
24
% java Euclid 314159 271828
1

```

这个程序给出了欧几里得算法的递归实现，以计算两个命令行参数的最大公约数。

欧几里得算法 两个正整数的最大公约数 (greatest common divisor, gcd) 是能同时整除这两个数字的最大整数。例如，102 和 68 的最大公约数是 34，因为 102 和 68 都是 34 的倍数，且没有大于 34 的整数能同时整除 102 和 68。在学习分数化简的时候，你可能已经学习过最大公约数了。例如，我们可以通过将分子和分母除以 34 (68 和 102 的最大公约数) 来简化 68/102 到 2/3。查找较大整数的最大公约数是许多商业应用中的重要问题，包括著名的 RSA 密码系统。

对于正整数 p 和 q ，计算其最大公约数时遵循以下定理：

267

如果 $p > q$ ， p 和 q 的最大公约数等于 q 和 $p \% q$ 的最大公约数。

为了证明上述定理，首先， p 和 q 的最大公约数等于 q 与 $p - q$ 的最大公约数，因为当且仅当一个数字能够同时整除 q 和 $p - q$ 时，它才能同时整除 p 和 q 。按照这样的推断过程， q 和 $p - 2q$ 、 q 和 $p - 3q$ 等应具有相同的最大公约数，并且计算 $p \% q$ 的一种方法是从 p 中不断减去 q 直到得到小于 q 的数字。

Euclid (程序 2.3.1) 中的静态方法 gcd() 是一个紧凑的递归函数，其归约步骤基于上述推断过程。基础步骤为：当 q 为 0 时， $\text{gcd}(p, 0) = p$ 。可以看到，每次递归调用时，只要还满足 $p \% q < q$ ，第二个参数的值就会严格递减，直到归约步骤收敛到基础步骤。如果 $p < q$ ，第一个递归调用会有效地切换了两个参数的顺序。实际上，第二个参数值在每一次递归调用时都至少减少 2 的倍数，所以参数值序列很快收敛到基础步骤 (参见练习 2.3.11)。这个计算最大公约数的递归解决方案被称为欧几里得算法，欧几里得算法是最古老的算法之一，迄今已有 2000 多年的历史。

```

gcd(1440, 408)
  gcd(408, 216)
    gcd(216, 192)
      gcd(192, 24)
        gcd(24, 0)
          return 24
        return 24
      return 24
    return 24
  return 24
gcd() 的函数调用跟踪

```

汉诺塔 每一次对递归话题的讨论，都不可避免地会提到古老的汉诺塔 (towers of Hanoi) 问题。所谓汉诺塔问题，就是假设有三根柱子和 n 个圆盘，圆盘套在柱子上。圆盘大小不同，最初所有的圆盘都套在一根柱子上，按照从大到小的顺序排列，底部是最大的圆盘 (圆盘 n)，顶部是最小的圆盘 (圆盘 1)。我们任务是将所有 n 个圆盘移动到另一个柱子

上，同时遵守以下规则：

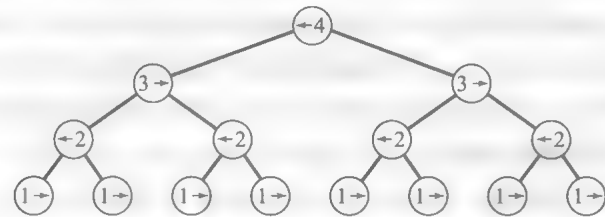
- 一次只移动一个圆盘。
- 不要把大圆盘放在小圆盘上。

传说当僧侣们在三根钻石针的 64 个金色圆盘上完成这项任务时，就是世界的末日。但是僧侣们怎样才能按照规则来完成这项任务呢？

为了解决这个问题，我们的目标是发布一系列移动圆盘的指令。我们假设柱子排列成一行，并且每个指令移动指定数量的圆盘向左或向右。如果圆盘位于最左侧柱子，向左移动意味着圆盘会回绕到最右侧的柱子上；如果圆盘位于最右侧柱子，则向右移动意味着圆盘会回绕到最左侧的柱子上。当圆盘全部在一侧时，有两个可执行的操作：向左或向右移动最小的圆盘；如果圆盘并不是都在一侧，则有三种可执行的操作：向左或向右移动最小的圆盘，或在其他两侧的柱子之间进行合理的移动。为了实现目标而精心计算每一步移动，这确实是一个挑战。借助递归，我们可以根据下面的计划来完成这个复杂的任务：首先我们将顶部的 $n-1$ 个圆盘移动到一个空柱子上，然后将底部最大的圆盘移动到另一个空柱子上（不影响较小的圆盘），然后我们通过将 $n-1$ 个圆盘移动到最大的圆盘所在的那根柱子来完成这项工作。

TowersOfHanoi（见程序 2.3.2）是这种递归策略的直接实现。它需要一个命令行参数 n ，然后可以输出 n 个汉诺塔圆盘的解决方案。其中，递归函数 moves() 会输出一系列移动指令，将圆盘向左（如果参数 left 为 true）或向右（如果 left 为 false）移动，并且移动的过程完全遵循上述方案的要求。

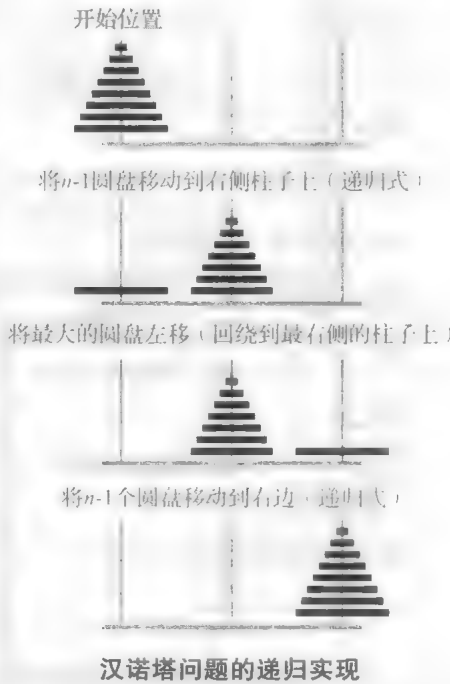
函数调用树 为了更好地理解具有多个递归的函数调用（如 TowersOfHanoi）的程序的行为，我们用函数调用树（function-call tree）来提供一种可视化的表示方法。具体来说，我们将每个方法调用视为树节点（tree node），用圆圈表示，每个圆圈里标注函数调用的参数的值。在每个树节点下面，我们绘制对应每个函数调用的子节点（从左到右），用线将树节点和子节点连接起来。该图包含了我们需要了解程序如何工作的所有信息，并包含每个函数调用的节点。



程序 TowersOfHanoi 中 moves(4, true) 的函数调用树

我们可以使用函数调用树来了解所有模块化程序的行为，它们对分析递归程序的行为大有帮助。例如，与 TowersOfHanoi 中的 move() 相对应的树很容易构建。首先绘制一个树

268



- 269 节点，用命令行参数的值标注。第一个参数是要移动的圆盘数（以及要实际移动的圆盘标签）；第二个参数是移动圆盘的方向。为了清楚起见，我们用向左或向右的箭头描述方向（布尔值），这样我们可以将值与方向联系起来。然后在下面绘制两个树节点，圆盘数减1，方向切换，重复上述过程，直到所有树节点的值为1，且它的下面没有子节点，这些节点对应于move()的调用不会再有后续的递归调用。

程序 2.3.2 汉诺塔问题

```
public class TowersOfHanoi
{
    public static void moves(int n, boolean left)
    {
        if (n == 0) return;
        moves(n-1, !left);
        if (left) StdOut.println(n + " left");
        else StdOut.println(n + " right");
        moves(n-1, !left);
    }
    public static void main(String[] args)
    {
        // 读取n，输出将n个圆盘向左移动的指令
        int n = Integer.parseInt(args[0]);
        moves(n, true);
    }
}
```

n	圆盘数量
left	圆盘移动方向

递归方法move()输出将n个圆盘向左（如果left为真）或向右（如果left为假）移动的指令。

```
% java TowersOfHanoi 1
1 left
% java TowersOfHanoi 2
1 right
2 left
1 right
% java TowersOfHanoi 3
1 left
2 right
1 left
3 left
1 left
2 right
1 left
```

```
% java TowersOfHanoi 4
1 right
2 left
1 right
3 right
1 right
2 left
1 right
4 left
1 right
2 left
1 right
3 right
1 right
2 left
1 right
```

270

将本节前面描述的函数调用树，与描述的相应函数调用轨迹进行比较，你会发现函数调用树只是函数调用轨迹的简化形式，而从左到右的节点标签即是解决问题的移动指令。

此外，当你学习树时，你可能会注意到几个特点，其中包括以下两个：

- 每隔一次即需要移动最小圆盘。
- 圆盘总是沿相同方向移动。

这些特点至关重要，因为不需要递归（甚至是计算机）它们就可以解决问题：每隔一次即需要移动最小圆盘（包括第一个和最后一个），那么，每个不涉及最小圆盘的移动是唯一有效的移动。我们可以证明，这种方法产生的结果与递归程序的结果相同。几个世纪以前（还没有计算机），也许僧侣们使用的就是这种方法。

函数调用树在理解递归中是非常重要的，因为其本身就是一个典型的递归对象。树作为抽象的数学模型，在许多应用中起着至关重要的作用。稍后在第4章中，我们将考虑使用树作为计算模型，来构建高效的数据结构。

指数级时间 使用递归的一个优点是，我们经常可以构建数学模型以推理或者证明递归程序中一些我们感兴趣的属性。例如，对于汉诺塔问题，我们就可以构建数学模型以估计到世界末日的时间（假设传说是真的）。这个问题很重要，不仅仅是因为它告诉我们世界的尽头是相当遥远的（即使传说是真实的），而且因为它提供给我们一些启发，帮助我们避免编写直到那时才能运行完的程序。

汉诺塔问题的数学模型很简单：如果我们将函数 $T(n)$ 定义为由 TowersOfHanoi 解决 n 个圆盘问题所需要移动的步数，则递归代码意味着 $T(n)$ 必须满足以下等式：

$$\begin{cases} T(n)=2T(n-1)+1, & \text{当 } n>1 \text{ 时} \\ T(n)=1, & \text{当 } n=1 \text{ 时} \end{cases}$$

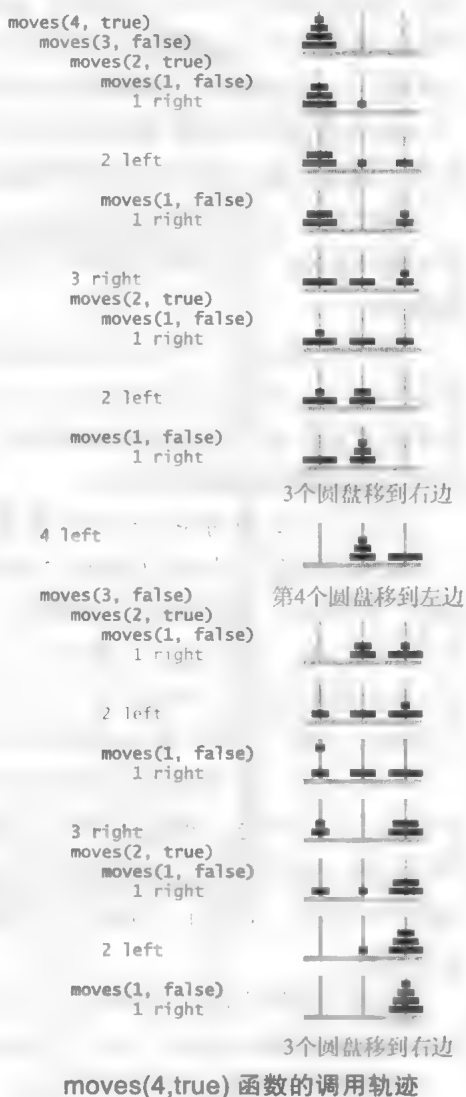
这个方程在离散数学中被认为是一种递推关系 (recurrence relation)。在研究递归程序时自然会出现递推关系，通过递推关系，我们可以推导出与 n 直接相关的、闭合形式的表达式。对于 $T(n)$ ，我们可以从前面的几个值 $T(1)=1$, $T(2)=3$, $T(3)=7$ 和 $T(4)=15$ ，猜测 $T(n)=2^n-1$ 。通过数学归纳法，可以证明该猜测为真：

- 基础步骤： $T(1)=2^1-1=1$
- 归纳步骤：如果 $T(n-1)=2^{n-1}-1$, $T(n)=2(2^{n-1}-1)+1=2^n-1$

因此，通过归纳，对于所有 $n>0$, $T(n)=2^n-1$ 。圆盘移动的最小可能步数也满足该递推关系（参见练习 2.3.11）。

已知 $T(n)$ 的值，我们可以估计执行所有移动所需的时间。如果僧侣以每秒移动 1 个圆盘的速度，则他们需要一个多星期才能完成 20 个圆盘问题，超过 34 年才能完成 30 个圆盘问题，超过 348 个世纪，才能完成 40 个圆盘的问题（假设他们没有犯错误）。而完成 64 个圆盘问题将需要 58 亿多个世纪。僧侣们不能借助程序 2.3.2，也就无法如此快速地移动圆盘或者快速找出接下来移动哪个圆盘，那么世界末日还要更遥远。

即便是计算机的计算速度也不能与指数级增长的运算量相匹配。比如说，每秒可以进行十亿次操作的计算机，仍然需要数百年的时间才能进行 2^{64} 次操作，目前还没有计算机能够执行 2^{1000} 次操作。这个教训是深刻的：通过递归，你可以轻松编写需要运行指数级时间的简短程序，但是当你尝试使用较大的 n 运行程序时，它们则无法完成运行。新手们经常对这个基本事实持怀疑态度，现在可以停下来做如下实验：将打印语句从程序 TowersOfHanoi 中删掉后运行， n 从 20 开始逐渐增加。很容易得出，每次将 n 的值增加 1，运行时间会加倍，



271

272

而你会很快失去耐心等待程序完成。对于某个 n 值，如果你需要等待一个小时；那么对于 $n+5$ ，你将多等一天；对于 $n+10$ ，你将多等一个月；对于 $n+20$ ，你将多等一个世纪，然而没有人能等那么久。计算机的速度没有快到可以运行你编写的每个 Java 程序，无论程序可能看起来多么简单！请注意那些需要指数级时间的程序。

我们常常希望能够预测我们程序的运行时间。在 4.1 节中，我们将讨论我们刚才用来估算其他程序运行时间的过程。

格雷码 汉诺塔问题不是游戏，它与用于数字编码和离散对象处理的基本算法密切相关。下面我们再来看一个例子——格雷码，一个广泛应用的数学抽象概念。

剧作家塞缪尔·贝克特 (Samuel Beckett) 的作品《等待戈多》闻名世界，他还写过一个戏剧，名为 *Quad*：在一个空白的舞台上，每次仅能有一个角色进入或者退出舞台，而每次留在舞台上的演员组合不能重复，贝克特应该如何为这个戏剧规划出舞台指令？

表示 n 个离散对象的子集的常用方法是使用一个 n 位字符串，每一位对应一个对象的状态。对于贝克特的这个问题，我们使用一个 4 位字符串，位数从右到左，位值为 1，表示该演员在舞台上，为 0 则表示不在。例如，字符串 0101 表示演员 3 和 1 在舞台上。这个表示方法快速地证明了一个基本事实： n 个对象有 2^n 个不同子集，Quad 有 4 个字符，所以有 $2^4=16$ 个不同集合。我们的任务是规划舞台指令。

一个 n 位二进制的格雷码包含了 2^n 个不同的二进制数，这些数字组成一个列表，列表中的每一个数字都与上一个数字仅有一位存在差异。格雷码可以直接应用于贝克特的问题，将一个二进制位的值从 0 改为 1 对应于一个演员进入舞台；从 1 更改为 0 对应演员退出舞台。

那我们如何生成格雷码呢？与解决汉诺塔问题时非常相似，我们仍可以使用递归方案。 n 位二进制的格雷码按递归方式定义如下：

- $(n-1)$ 位的格雷码，在前面加一位 0。
- 然后是 $(n-1)$ 位的格雷码逆序排列，在前面加一位 1。

0 位的格雷码被定义为空，因此 1 位的格雷码是 0 和 1。根据这个递归定义，我们可以通过数学归纳法证明，这样的编码方式符合格雷码的要求：任意相邻的两行代码只有一位代码不同。证明的过程非常容易，根据归纳假设，显然前半部分和后半部分都可以使命题成立，而上半部分的最后一行编码和下部分的第一行编码只有第一位不同，因此整体命题为真。

经过仔细思考，我们根据递归定义可以写出程序 Beckett (程序 2.3.3)，并输出贝克特需要的舞台指令。这个程序与程序 TowersOfHanoi 非常相似。的确，除命名之外，唯一的区别是递归调用中第二个参数的值！

与 TowersOfHanoi 的指令一样，Beckett 出入舞台的

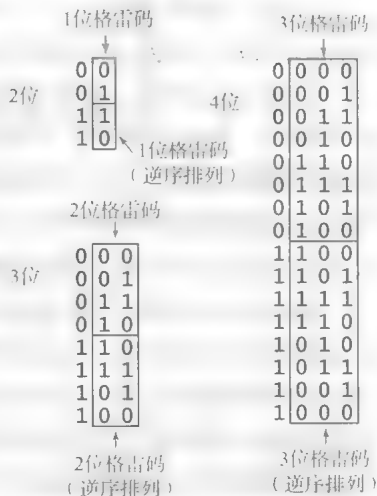
(30, 2^{30})

(20, 2^{20})

指数级增长

编码	子集	指令
0 0 0 0	空	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

格雷码示意图



2 位、3 位、4 位格雷码

enter 和 exit 指令是多余的，因为只有当演员在舞台上时才会执行 exit 操作，只有当演员不在舞台上时才执行 enter 操作。事实上，Beckett 和 TowersOfHanoi 都与我们在第 1 章学习的 ruler 函数（程序 1.2.1）有很密切的关系。去掉打印指令之后，它们都是一个简单的递归函数，可以用来在 Ruler 程序中根据命令行参数计算出标尺上的刻度线长度。

格雷码有许多应用，包括从模 - 数转换器到实验设计。它们已经用于脉冲代码通信、逻辑电路的最小化和超立体架构，甚至被建议用于组织图书馆书架上的书籍。

274

程序 2.3.3 格雷码

```
public class Beckett
{
    public static void moves(int n, boolean enter)
    {
        if (n == 0) return;
        moves(n-1, true);
        if (enter) StdOut.println("enter " + n);
        else StdOut.println("exit " + n);
        moves(n-1, false);
    }

    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        moves(n, true);
    }
}
```

n	演员人数
enter	舞台指令

这个递归程序输出贝克特所需的舞台指令（二进制表示的格雷码位的变化表示舞台的进入和退出指令）。ruler 函数精确地描述了位的变化，并且（当然）也描述了每个演员交替进入和退出的指令。

```
% java Beckett 1
enter 1
% java Beckett 2
enter 1
enter 2
exit 1
% java Beckett 3
enter 1
enter 2
exit 1
enter 3
enter 1
exit 2
exit 1
```

```
% java Beckett 4
enter 1
enter 2
exit 1
enter 3
enter 1
exit 2
exit 1
enter 4
enter 1
enter 2
exit 1
exit 3
enter 1
exit 2
exit 1
```

275

递归图形 简单的递归绘制方案可以绘制复杂的图像。递归制图不仅涉及许多应用，还为更好地理解递归函数的属性提供了一个有趣的平台，因为借此我们可以观察递归图像的生成过程。

下面我们给出第一个简单的例子——Htree（程序 2.3.4），给定一个命令行参数 n ，绘制一个 n 阶 H 树。 n 阶 H 树的定义如下：

- 1) 基础步骤：当 $n=0$ 时，不绘制任何东西。
- 2) 归约步骤：在单位正方形内绘制。
 - 字母 H 形状的四条线段

- 分别以字母 H 的 4 个顶端为中心绘制 4 棵 $n-1$ 阶的 H 树，附加条件是 $n-1$ 阶的 H 树的大小是刚刚绘制的一半。

像这样的图有很多实际应用。例如，假设一个有线电视公司要将自己的线缆连接到其所在地区的所有家庭，合理的策略是使用 H 树将信号发送到分布在整个区域的若干个中心节点，然后将线缆从距离最近的中心节点连接到每个家庭。计算机设计人员想要在整个集成电路芯片中分配电源或信号时，面临的是同样的问题。

虽然每次绘制都是在指定大小的区域内完成的，但是 H 树一定会呈指数级增长。 n 阶 H 树会连接 4^n 个中心。当 $n=10$ 时，你将会绘制一百多万条线；当 $n=15$ ，你将会绘制十亿多条线；当 $n=30$ 时，你的程序运行不完。

如果你花点时间在计算机上运行绘制一个大概需要一分钟左右的 H 树，你只需通过观察绘制进度，就可以深入了解递归程序的性质，因为你可以看到每个 H 字母的出现过程以及形成 H 树的过程。你可以尝试一个更有启发性的练习，可以改变 `draw()` 函数和 `StdDraw.line()` 函数的调用顺序，但是不管改变后的顺序如何，得到的图像总是一样的，而改变的只是线条出现在绘图中的顺序（见练习 2.3.14）。

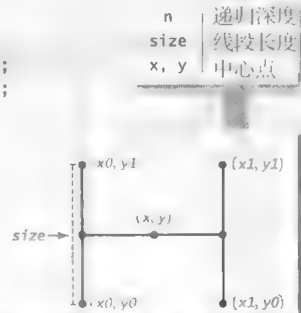


[276]

程序2.3.4 递归图形

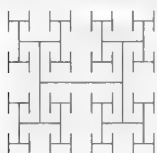
```
public class Htree
{
    public static void draw(int n, double size, double x, double y)
    { // 在 (x,y) 点绘制H树
        // 递归深度为n, 给定线段长度 size
        if (n == 0) return;
        double x0 = x - size/2, x1 = x + size/2;
        double y0 = y - size/2, y1 = y + size/2;
        StdDraw.line(x0, y, x1, y);
        StdDraw.line(x0, y0, x0, y1);
        StdDraw.line(x1, y0, x1, y1);
        draw(n-1, size/2, x0, y0);
        draw(n-1, size/2, x0, y1);
        draw(n-1, size/2, x1, y0);
        draw(n-1, size/2, x1, y1);
    }

    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        draw(n, 0.5, 0.5, 0.5);
    }
}
```

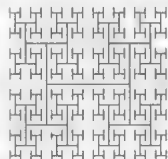


函数 `draw()` 以 (x, y) 为中心，依照字母 H 的形状，绘制三条线段，每条线段长度为 `size`。然后，它在四个顶点调用自身，每次调用中 `size` 减半。整个函数通过正整数 `n` 来控制递归的深度。

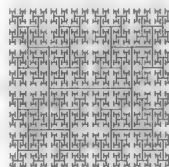
% java Htree 3



% java Htree 4



% java Htree 5

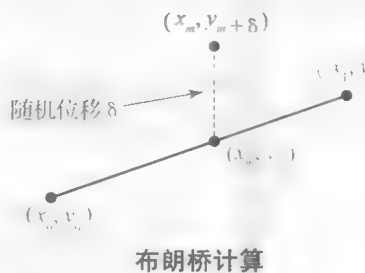


布朗桥 H 树是分形 (fractal) 的一个简单示例：分形图形是一个几何形状，它可以被分为几个部分，每个部分都是 (或近似于) 原始版本的尺寸缩小版。科学家、数学家和程序员从许多不同的角度来研究分形。递归程序很容易生成分形图形。本书有几个分形的例子，如 IFS (见程序 2.2.3)。

分形图形的研究在艺术表达、经济分析和科学发现中起着重要而持久的作用。艺术家和科学家使用分形来构建自然界中出现的复杂形状的紧凑模型，并取代了常规的几何形状模型，如云、植物、山脉、河床、人体皮肤等使用的模型描述都是分形图形。经济学家则使用分形来模拟经济指标的函数图。

分形布朗运动 (fractional Brownian motion) 是一个数学模型，它为许多自然存在的凹凸不平的形状创建逼真的分形模型。它用于研究金融和许多自然现象，包括洋流和神经细胞膜。精确计算分形模型可能是一个困难的挑战，但使用递归程序计算近似值并不难。

Brownian (程序 2.3.5) 能够生成一个函数图，它是分形布朗运动的近似示例，通常被称作布朗桥 (Brownian bridge)。你可以将该图形视作在两个点 (x_0, y_0) 和 (x_1, y_1) 的连线间随机游走，并由几个参数控制。程序的实现基于中点位移法 (midpoint displacement method)，该方法是在 x 的区间 $[x_0, x_1]$ 内递归式地绘图。基本步骤是，当两个端点间的距离小于给定的最小值时，直接绘制连接两个端点的线段。归约步骤是将间隔分为两半，步骤如下：



- 计算间隔的中点 (x_m, y_m) 。
- 向 y 坐标的中点 y_m 添加一个随机值 δ ， δ 满足高斯分布，其中高斯分布的均值为 0，方差由程序设定。
- 在子区间上递归，将方差除以给定的比例系数 s 。

曲线的形状由两个参数控制：波动率 (volatility, 方差的初始值) 控制函数曲线偏离原来两点之间直接相连的线段的距离，Hurst 指数 (Hurst exponent) 控制曲线的平滑度。我们用 H 表示 Hurst 指数，并在每次递归时将方差除以 2^{2H} 。当 H 是 $1/2$ (每次递归减半) 时，曲线是一个布朗桥，即赌徒破产问题的连续版本 (见程序 1.3.8)。当 $0 < H < 1/2$ 时，偏离趋于增加，使得程序绘制出较粗糙的曲线。当 $2 > H > 1/2$ 时，偏离趋于减小，使得绘出的曲线更平滑。值 $2-H$ 被称为曲线的分形维度 (fractal dimension)。

间隔的波动率和初始端点与缩放和位置有关。**Brownian** 中的测试客户程序 `main()` 允许我们改变 Hurst 指数。如果值大于 $1/2$ ，你绘制的图像会像山地景观的地平线；如果值小于 $1/2$ ，你会得到与股票指数相似的图。

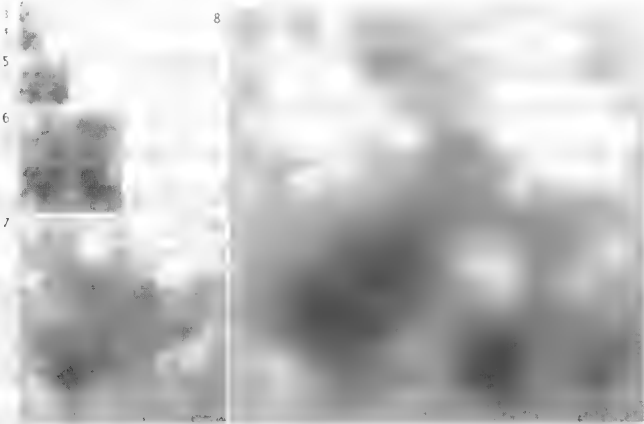
将中点位移法扩展到二维，产生的分形图像被称为等离子体云 (plasma cloud)。为了绘制矩形等离子体云，我们使用递归，其中基础步骤是绘制给定颜色的矩形，归约步骤是在四个象限中绘制等离子体云，颜色受到随机高斯分布的平均值的影响。使用与 **Brownian** 相同的波动率和平滑度，我们可以生成非常逼真的合成云。我们可以使用相同的代码来生成合成地形，用颜色值来表示不同海拔。这种方案的变种广泛应用于娱乐业，特别是用于为电影和游戏创造背景风光。

程序2.3.5 布朗桥

```
public class Brownian
{
    public static void curve(double x0, double y0,
                             double x1, double y1,
                             double var, double s)
    {
        if (x1 - x0 < 0.01)
        {
            StdDraw.line(x0, y0, x1, y1);
            return;
        }
        double xm = (x0 + x1) / 2;
        double ym = (y0 + y1) / 2;
        double delta = StdRandom.gaussian(0, Math.sqrt(var));
        curve(x0, y0, xm, ym+delta, var/s, s);
        curve(xm, ym+delta, x1, y1, var/s, s);
    }
    public static void main(String[] args)
    {
        double hurst = Double.parseDouble(args[0]);
        double s = Math.pow(2, 2*hurst);
        curve(0, 0.5, 1.0, 0.5, 0.01, s);
    }
}
```

x0, y0	左端点
x1, y1	右端点
xm, ym	中点
delta	偏离
var	方差
hurst	Hurst指数

通过向递归程序添加一个小的随机数（符合高斯分布），我们将绘制一个分形曲线；若不在递归程序中增加随机数，我们将绘制一条直线。称为Hurst指数的命令行参数hurst控制曲线的平滑度。



等离子体云

279
280

递归的陷阱 现在，你相信递归可以帮助你编写紧凑优雅的程序。当你开始编写自己的递归程序时，你需要注意可能出现的几个常见错误。我们已经详细讨论了其中的一个（程序

的运行时间可能呈指数增长)。一旦发现这些问题，一般不难克服，但在编写时你要非常小心地避免这些问题。

缺少基础步骤。思考以下递归函数，它应该计算谐波数，但是缺少了基础步骤：

```
public static double harmonic(int n)
{
    return harmonic(n-1) + 1.0/n;
}
```

如果你运行调用此函数的客户程序，它将重复调用自身，永远不会返回，因此你的程序将永远不会终止。你可能已经碰到过死循环——你调用了你的程序后，但什么都没有发生（或者程序不停地输出）。然而，随着无限递归，结果会不一样，因为系统会跟踪每个递归调用的信息（使用一个称为栈的数据结构，我们将在 4.3 中讨论使用的机制），最终会用尽内存。最终，Java 在运行时报告 `StackOverflowError`。当你编写递归程序时，你需要通过基于数学归纳的非正式论证来说服自己，以确保程序能够达到预期的效果。这样做可能会发现遗失了基础步骤。

不能保证收敛。另一个常见的问题是，在递归函数中包含一个递归调用来解决一个子问题，而这个子问题的规模并不小于原始问题。例如，对于参数的任何值（除了 1），以下方法进入无限递归循环，因为参数值序列不会收敛到基础步骤：

```
public static double harmonic(int n)
{
    if (n == 1) return 1.0;
    return harmonic(n) + 1.0/n;
}
```

281

这个例子中的错误很容易发现，但同样是这类问题，如果参数中有细微的差别，可能就不那么好辨别了。你可以在本节的练习中找到几个例子。

内存要求过量。如果一个函数在返回之前递归调用多次，则 Java 所需的递归调用的内存可能无法满足需求，从而导致报告 `StackOverflowError`。要想知道涉及了多少内存，可以运行一组实验：使用递归函数来计算谐波数，并不断增加 n 的值。

```
public static double harmonic(int n)
{
    if (n == 1) return 1.0;
    return harmonic(n-1) + 1.0/n;
}
```

程序报告 `StackOverflowError` 时，你对 Java 使用多少内存来实现递归就有了一定的了解。相比之下，运行程序 1.3.5 计算 H_n ，当 n 很大时，也只使用很小的内存。

重复计算过量。一个函数的过量重复计算，可能导致函数运行时间呈指数级增长。一旦了解了这一点，在编写简单的递归程序以解决简单问题时，就会更加谨慎。即使在最简单的递归函数中，这种情况也是存在的，你需要尽量避免它。例如，斐波那契数列

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,...

被定义为：当 $n \geq 2$ 时，循环 $F_n = F_{n-1} + F_{n-2}$ ，并且 $F_0 = 0$, $F_1 = 1$ 。斐波那契序列具有许多有趣的特性，并出现在很多应用中。一个新手程序员可能会通过如下递归函数来计算斐波那契数列：

```
// Warning: 这个函数非常低效
public static long fibonacci(int n)
{
```

```

    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}

```

[282]

但是，这个函数是非常没有效率的！新手程序员通常不愿相信这个事实，期望计算机具有足够的速度可以运行这样的代码以解决问题。下面我们用这个函数来计算 `fibonacci(50)`，看看计算机是否具备足够的速度。首先思考计算 `fibonacci(8)=21` 时，函数需要先计算 `fibonacci(7)=13` 和 `fibonacci(6)=8`。计算 `fibonacci(7)` 时，递归计算 `fibonacci(6)=8` 和 `fibonacci(5)=5`，事情很快变得越来越糟，因为计算 `fibonacci(6)`，计算机忽略了已经计算过 `fibonacci(5)` 等等，以此类推，这个程序在计算 `fibonacci(n)` 时计算 `fibonacci(1)` 的次数正好是 F_n （见练习 2.3.12），重复计算使得错误呈指数级增长。例如，计算 `fibonacci(200)`，函数要计算 $F_{200} > 10^{43}$ 次 `fibonacci(1)`。很难想象有哪台计算机可以完成这么多的计算。注意那些需要指数级运行时间的程序。许多计算任务直接想到的最简单的实现方法都属于这一类，当心不要陷入试图实施和运行它们的陷阱。

接下来，我们会学习一种避免此类错误的系统技术，称为动态编程（dynamic programming）。动态编程通过保存已经计算过的值以供后续使用，替代了不断重复的计算，以避免重复计算的过量。

[283]

动态编程 动态编程是实现递归程序常用的一种策略，它为众多问题提供了有效的解决方案。其基本思想是将复杂问题递归地分解成多个简单的子问题，存储每个子问题的答案，并最终使用存储的答案来解决原始问题。每个子问题只会被解决一次（而不是一次又一次），这种技术避免了潜在的运行时间指数级爆炸式的增长。

例如，如果我们的原始问题是计算第 n 个斐波那契数，则自然地划分为 $n+1$ 个子问题，其中子问题 i 是计算第 i 个斐波那契数，其中 $0 \leq i \leq n$ 。如果我们已经知道了较小子问题的解决方案，我们就可以轻松地解决子问题，在这个例子中，如果解决了子问题 $i-1$ 和 $i-2$ ，那就很容易解决第 i 个子问题。此外，求解原始问题也变成了解决其中一个子问题——第 n 个子问题。

自上而下的动态规划。在自上而下的动态编程中，我们存储或缓存我们解决的每个子问题的结果，以便下次我们需要解决相同的子问题时，我们可以使用缓存的值，而不是从头开始解决子问题。对于斐波那契数列的例子，我们使用数组 `f[]` 存储已经计算过的斐波那契数。在 Java 中，我们通过使用在任何方法之外声明的静态变量（也称为类变量或全局变量）来实现此操作。这允许我们将信息从一个函数调用保存到下一个。

```

fibonacci(8)
  fibonacci(7)
    fibonacci(6)
      fibonacci(5)
        fibonacci(4)
          fibonacci(3)
            fibonacci(2)
              fibonacci(1)
                return 1
              fibonacci(0)
                return 0
            return 1
          fibonacci(1)
            return 1
          return 2
        fibonacci(2)
          fibonacci(1)
            return 1
          fibonacci(0)
            return 0
          return 1
        return 3
      fibonacci(3)
        fibonacci(2)
          fibonacci(1)
            return 1
          fibonacci(0)
            return 0
          return 1
        fibonacci(1)
          return 1
        return 2
      return 5
    fibonacci(4)
      fibonacci(3)
        fibonacci(2)
          .
          .
          .

```

计算斐波那契数列的错误方法

```

public class TopDownFibonacci
{
    private static long[] f = new long[92];

    public static long fibonacci(int n)
    {
        if (n == 0) return 0;
        if (n == 1) return 1;
        if (f[n] > 0) return f[n];
        f[n] = fibonacci(n-1) + fibonacci(n-2);
        return f[n];
    }
}

```

缓存的值

静态变量（在所有方法之外声明）

返回缓存值（如果已经计算过）

计算当前值并保存

用于计算斐波那契数的自上而下的动态编程方法

自上而下的动态编程也被称为记忆（memorization），因为它通过记住函数调用的结果避免了重复的工作。

284

自底向上的动态编程。在自底向上的动态编程中，我们从“最简单”的子问题开始，对所有子问题进行计算，逐渐建立起对越来越复杂的子问题的解决方案。为了完成自底向上的动态编程，我们必须对子问题进行排序，以便每个后续的子问题可以通过先前已经解决的子问题相互组合来解决。对于斐波那契数列的例子，这很容易：按照 0、1、2 等的顺序解决子问题。当我们需要解决子问题 i 时，我们已经解决了所有更小的子问题，也就是子问题 $i-1$ 和 $i-2$ 。

```

public static long fibonacci(int n)
{
    int[] f = new int[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}

```

当子问题的排序是明确的，同时空间可用于存储所有的解决方案时，自底向上的动态编程是一个非常有效的方法。

接下来，我们学习动态编程更复杂的应用，其中解决子问题的顺序不是那么明确。与计算斐波那契数字的问题不同，这个问题若没有递归式思考，以及自底向上的动态编程方式，将很难得到解决。

最长公共子序列问题。下面我们来讨论一个字符串处理的基础问题，它也常应用于生物学或其他领域。假设有两个给定的字符串 x 和 y ，我们想比较它们的相似度。常见的类似问题包括比较两个 DNA 序列的同源性、两个英文单词拼写的相似性或两个 Java 代码文件的重复度等。相似度的度量方法是计算最长公共子序列（Longest Common Subsequence, LCS）的长度。如果从 x 中删除一些字符，并从 y 中删除一些字符，使得生成的两个字符串相等，则我们将生成的字符串称为公共子序列（common subsequence）。LCS 问题是找到两个字符串的尽可能长的公共子序列。例如，GGCACCACG 和 ACGGCGGATACG 的 LCS 是字符串 GGCAACG，它的长度为 7。

285

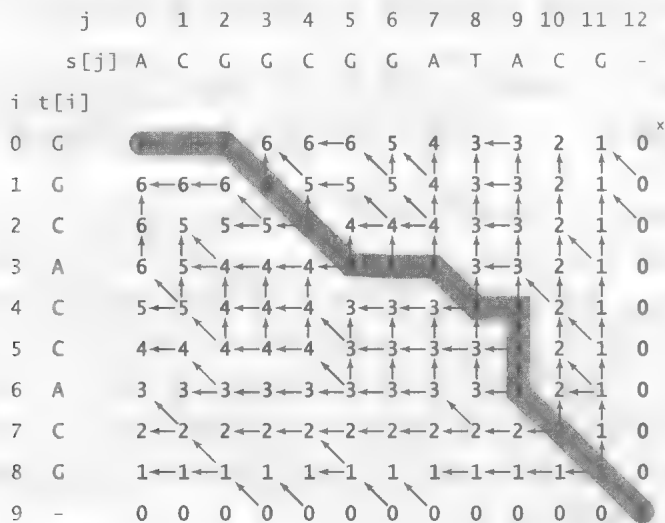
计算 LCS 的算法很多用于数据比较程序，如 UNIX 中的 diff 命令，已经被使用了几十年，程序员可以通过这个命令找到文本文件中的差异和相似之处。类似的算法在科学应用中扮演重要角色，如计算生物学中的史密斯-沃特曼算法（Smith-Waterman algorithm）和数字通信理论中的维特比算法（Viterbi algorithm）。

LCS 的递推。现在我们描述一个递归公式，帮助我们找到两个给定的字符串 s 和 t 的 LSC。令 m 和 n 分别为 s 和 t 的长度，我们使用符号 $s[i..m]$ 表示从索引 i 开始的 s 的后半部分，称为 s 的后缀， $t[j..n]$ 表示从索引 j 开始的 t 的后缀。一种情况下，如果 s 和 t 以相同的字符开始，则 s 和 t 的 LCS 就包含第一个字符。因此，我们的问题可以化简为找到后缀 $s[1..m]$ 和 $t[1..n]$ 的 LCS。另一种情况下，如果 s 和 t 以不同的字符开始，则两个字符串的首字符都不能成为公共子序列的一部分，所以我们可以放心地放弃其中一个字符。在这种情况下，问题都会化简为找到 $s[0..m]$ 和 $t[1..n]$ 的 LCS 或 $s[1..m]$ 和 $t[0..n]$ 的 LCS，并取其中较大的一个^①（原来的问题变成了两个子问题，但是两个子问题的规模都在严格缩小——译者注）。不失一般性地，我们让 $\text{opt}[i][j]$ 表示后缀 $s[i..m]$ 和 $t[j..n]$ 的 LCS 的长度，则以下递推表达式表达了 $\text{opt}[i][j]$ 的计算规则：

$$\text{opt}[i][j] = \begin{cases} 0 & , i = m \text{ 或 } j = n \\ \text{opt}[i+1, j+1] + 1 & , s[i] = t[j] \\ \max(\text{opt}[i, j+1], \text{opt}[i+1, j]) & \text{其他} \end{cases}$$

动态编程解决方案。LongestCommonSubsequence（程序 2.3.6）采用自底向上的动态编程方法解决这一递推问题。我们维护二维数组 $\text{opt}[i][j]$ ，存储后缀 $s[i..m]$ 和 $t[j..n]$ 的 LCS 的长度。最初，底行 ($i=m$) 和右列 ($j=n$) 为 0，这些是初始值。从递推计算公式的定义，我们可以清楚地看出后续计算的顺序：我们从 $\text{opt}[m][n]=1$ 开始，然后只要逐一减少 i 或 j 或两者同时减少，就可以逐步计算出需要计算的 $\text{opt}[i][j]$ 。每当计算 $\text{opt}[i][j]$ 时，它所需要的是下标大于 i 或 j 或同时大于两者的 $\text{opt}[][]$ 中的元素，而那些元素已经计算出来了。程序 2.3.6 中的方法 `lcs()` 通过按行从下到上 (i 从 $m-1$ 到 0)，每行从右到左 (j 从 $n-1$ 至 0) 的顺序填充 $\text{opt}[][]$ 中的元素。另外一种替代方案是按列从右到左，并每列从下到上的顺序进行逐个求值并填充。在下图中，每个元素都有一个或两个指向它的蓝色箭头，用于表示计算它时所用的值（当按照公式中求最大值进行计算时，就会显示两个箭头）。

[286]



字符串 GGCACCACG 和 ACGGCGGATACG 的最长公共子序列

① 这个地方原书是 x, y ，应该是错了。——译者注

② 原书有误。——译者注

③ 这里原书是 m ，应该是写错了。——译者注

程序2.3.6 最长公共子序列

```
public class LongestCommonSubsequence
{
    public static String lcs(String s, String t)
    {
        // 计算所有子问题的LCS长度
        int m = s.length(), n = t.length();
        int[][] opt = new int[m+1][n+1];
        for (int i = m-1; i >= 0; i--)
            for (int j = n-1; j >= 0; j--)
                if (s.charAt(i) == t.charAt(j))
                    opt[i][j] = opt[i+1][j+1] + 1;
                else
                    opt[i][j] = Math.max(opt[i+1][j], opt[i][j+1]);
        // 返回最长公共子序列本身
        String lcs = "";
        int i = 0, j = 0;
        while(i < m && j < n)
        {
            if (s.charAt(i) == t.charAt(j))
            {
                lcs += s.charAt(i);
                i++;
                j++;
            }
            else if (opt[i+1][j] >= opt[i][j+1]) i++;
            else j++;
        }
        return lcs;
    }

    public static void main(String[] args)
    {
        StdOut.println(lcs(args[0], args[1]));
    }
}
```

s, t	两个字符串
m, n	两个字符串的长度
opt[i][j]	x[i..m] 和 y[j..n]的 LCS的长度
lcs	最长公共子序列

函数lcs()使用自底向上的动态编程方法，计算并返回两个字符串x和y的LCS。方法调用s.charAt(i)返回字符串s的字符i。

```
% java LongestCommonSubsequence GGCACCACG ACGCGGATACG
GGCAACG
```

最后的问题是如何返回最长公共子序列本身，而不仅仅只返回它的长度。解决问题的关键在于回溯（backward）动态编程算法的步骤，重新发现 opt[0][0] 到 opt[m][n] 的选择路径（上图中以灰色突出显示）。为了找出我们如何确定 opt[i][j] 的选择，需要考虑三种可能性：

- 字符 s[i] 等于 t[j]。在这种情况下，我们一定可以得出 opt[i][j]=opt[i+1][j+1]+1，LCS 中的下一个字符是 s[i]（或 t[j]），所以我们在 LCS 中包括字符 s[i]（或 t[j]），并继续从 opt[i+1][j+1] 追溯。
- LCS 不包含 s[i]。在这种情况下，opt[i][j]=opt[i+1][j]，我们继续追溯 opt[i+1][j]。
- LCS 不包含 t[j]。在这种情况下，opt[i][j]=opt[i][j+1]，我们继续从 opt[i][j+1] 追溯。

我们从 opt[0][0] 开始追踪并一直迭代直到 opt[m][n]。在追溯的每个步骤中，i 增加 1 或者 j 增加 1（或两者同时增加 1），我们可以用一个 while 循环实现该过程，这个循环将在最多 m+n 次迭代之后终止。

动态编程是基本算法的一种设计范式，与递归密切相关。如果你稍后继续学习算法或运算研究课程，那么你一定学会更多相关的知识。递归是计算的基础，避免重复计算当然也是计算的一个基础。并不是所有的问题都可以直接转换为递归公式，也不是所有的递归公式

都包含一种可以避免重复计算的计算顺序——有时候你会一下子同时解决两个问题，既找到递归公式又给出避免重复计算的方法，正如你刚刚看到的 LCS 问题一样，第一次碰到这样的情况时或许会让你觉得不可思议。

总结 不使用递归的程序员会失去两个优势：首先，递归可以实现复杂问题的紧凑解决方案；其次，递归解决方案可以保证程序按照计划预期运行。在早期的计算中，递归程序因相关的开销过高而在一些系统中禁止使用，因此许多程序员会尽量避免使用递归。但是，在像 Java 这样的现代系统中，使用递归通常会是一个明智的选择。

递归函数真正展现了精心阐述抽象概念的力量。虽然一个函数能够调用自身这一概念起初对许多人来说似乎是荒唐的，但我们讨论的许多例子强有力地说明，掌握递归对于理解和利用计算以及理解计算模型在研究中的作用至关重要。

递归给我们强化了一个理念，在编程时需要证明一个程序可以按照预期运行。递归与数学归纳之间的自然联系至关重要。对于日常编程，我们对程序正确性的关注主要集中于如何节省用于跟踪错误的时间和精力。在现代应用中，安全和隐私问题使正确性成为编程至关重要的组成部分。如果程序员不能确定应用程序是否按预期方式工作，那么如何保障个人数据的私密性和安全性呢？

自从 20 世纪后半叶以来，计算机在日常生活中逐渐发挥核心作用，而在开发这些宏大的计算基础设施的过程中，递归就是程序设计模块中最后关键一环。程序由库和函数构成，而函数由各类语句组成，包括基本类型数据、条件、循环和函数调用（包括递归操作）执行语句等。这些方法和工具可以解决各种重要的问题。在下一节中，我们将强调这一点，并在一个大型应用程序的上下文中回顾这些概念。在第 3 章和第 4 章中，我们将扩展这些概念，让你领略更广泛的编程模式，从而深入理解现代计算机世界。

[289]

问答环节

问：是否存在只能通过迭代来解决的问题？

答：不，所有循环都可以由递归函数替代，尽管递归版本可能需要过多的内存。

问：是否存在只能通过递归来解决的问题？

答：不，所有递归函数都可以被对应的迭代所代替。在 4.3 节中，我们将看到编译器如果使用一种被称为栈的数据结构来生成函数调用过程的代码。

问：对于递归和迭代，我应该选择哪种？

答：哪一种能让你写出更简单、更容易理解或更高效的代码，就选择哪一种。

问：关于递归代码，我已经注意到了它会消耗更多的空间，还有重复计算的问题，还有其他需要关注的吗？

答：在递归代码中创建数组需要格外谨慎。使用的空间量可能会急剧增加，同时内存管理所需的时间也会急剧增加。

[290]

练习

2.3.1 如果你使用负值 n 调用 `factorial()`，会发生什么？或者 n 较大时，例如 35，会发生什么？

2.3.2 编写一个递归函数，它以整数 n 为参数，返回 $\ln(n!)$ 。

2.3.3 调用 `ex233(6)` 后，打印出的整数序列是什么？

```
public static void ex233(int n)
{
    if (n <= 0) return;
    StdOut.println(n);
    ex233(n-2);
    ex233(n-3);
    StdOut.println(n);
}
```

2.3.4 给出调用 ex234(6) 的结果值:

```
public static String ex234(int n)
{
    if (n <= 0) return "";
    return ex234(n-3) + n + ex234(n-2) + n;
}
```

2.3.5 指出以下递归函数的错误:

```
public static String ex235(int n)
{
    String s = ex235(n-3) + n + ex235(n-2) + n;
    if (n <= 0) return "";
    return s;
}
```

参考答案: 因为基础步骤出现在归纳步骤之后, 所以该函数不会收敛。调用 ex235(3) 将导致调用 ex235(0)、ex235(-3)、ex235(-6) 等, 以此类推, 直到报告错误: StackOverflowError。

291

2.3.6 给了 4 个正数 a、b、c 和 d, 试着计算 gcd(gcd(a,b)、gcd(c,d))。

2.3.7 假设 p 是整数和 q 是除数, 解释下列欧几里得函数的作用:

```
public static boolean gcdlike(int p, int q)
{
    if (q == 0) return (p == 1);
    return gcdlike(q, p % q);
}
```

2.3.8 思考以下递归函数:

```
public static int mystery(int a, int b)
{
    if (b == 0) return 0;
    if (b % 2 == 0) return mystery(a+a, b/2);
    return mystery(a+a, b/2) + a;
}
```

mystery(2,25) 和 mystery(3,11) 的结果值是什么? 给定正整数 a 和 b, 计算 mystery(a, b) 的结果值。用 “*” 替换 “+”, 用返回 1 代替返回 0, 再次回答相同的问题。

2.3.9 编写递归程序 Ruler, 用 StdDraw 绘制标尺中更细小的刻度, 如程序 1.2.1 所示。

2.3.10 假设 $T(1)=1$, n 是 2 的整数幂, 编写程序实现如下递归关系。

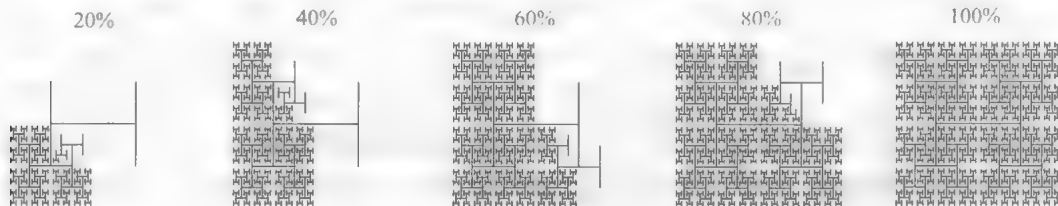
- $T(n)=T(n/2)+1$
- $T(n)=2T(n/2)+1$
- $T(n)=2T(n/2)+n$
- $T(n)=4T(n/2)+3$

2.3.11 通过归纳法证明, 递归方法得到的解法是解决汉诺塔问题的最小移动次数的解决方案。

292

2.3.12 通过归纳法证明, 当使用书中给出的递归程序计算 fibonacci(n) 时, 对 fibonacci(1) 的递归调用次数为 F_n 。

- 2.3.13 证明递归调用 `gcd()` 时，第二个参数每次调用时递减的速率至少为 2 的倍数。然后证明：
`gcd(p,q)` 最多使用 $2\log_2 n + 1$ 次递归调用，其中 n 是 p 和 q 中的较大值。
- 2.3.14 修改 Htree 程序（程序 2.3.4）为 H 树的图形设置动画。然后，重新排列递归调用（以及基础步骤）的顺序，查看生成的动画，并解释你得到的每个结果。



293

创新练习

- 2.3.15 二进制表示。写一个程序，以一个正整数 n （十进制）作为命令行参数，打印出该正整数的二进制表示形式。回顾在程序 1.3.7 中我们使用了减去 2 的幂方的方法。现在，我们使用下面更简单的方法：重复地将 n 除以 2 并反向输出每次计算获得的余数。首先写一个 `while` 循环来完成这样的计算，并且按顺序打印出每次计算获得的余数（此时的顺序是错误的），然后将顺序翻转，打印出余数正确的顺序。
- 2.3.16 A4 纸。在 ISO 标准下纸的宽高比是 $\sqrt{2}:1$ 。A0 纸的面积是 1 平方米；A1 纸是将 A0 纸沿垂直方向一分为二；A2 纸是 A1 纸沿水平方向一分为二；以此类推。写一个程序使得以一个整数 n 作为命令行参数并使用 `StdDraw` 来展示如何将一张 A0 纸切成 2^n 张。
- 2.3.17 排列。写一个程序 `Permutations`，以一个整数 n 作为命令行参数并打印出所有的 n 个字母（从字母 a 开始）的 $n!$ 种排列（假设 n 不大于 26）。 n 个元素的一个排列是这些元素 $n!$ 种可能的排列顺序之一。举个例子，当 $n=3$ 时，你应该获得下面的输出（不用考虑枚举所有排列时输出的顺序）：

```
bca cba cab acb bac abc
```

- 2.3.18 元素个数为 k 的排列。修改上个例子中 `Permutations` 程序使得它可以以两个参数 n 和 k 作为命令行参数，并打印出所有刚好包含这 n 个元素中 k 个元素的 $P(n,k)=n!/(n-k)!$ 种排列。下面是当 $k=2, n=4$ 时的预期输出（同样不用考虑输出的顺序）：

```
ab ac ad ba bc bd ca cb cd da db dc
```

- 2.3.19 组合。写一个程序 `Combinations`，它以一个整数 n 作为命令行参数并打印出所有的 2^n 个任意长度的组合。一个组合是 n 个元素的一个子集，不要考虑顺序。举个例子，当 $n=3$ 时，你应该得到下面的输出：

```
a ab abc ac b bc c
```

294

注意你的程序需要打印出一个空字符串（长度为 0 的子集）。

- 2.3.20 元素个数为 k 的组合。修改上个例子中的 `Combinations` 程序使得它以两个整数 n 和 k 作为命令行参数，并打印出所有的 $C(n,k)=n!/(k!(n-k)!)$ 个长度为 k 的组合。举个例子，当 $n=5, k=3$ 时，你应该得到下面的输出：

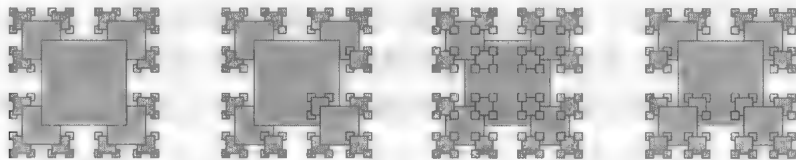
```
abc abd abe acd ace ade bcd bce bde cde
```

- 2.3.21 汉明距离。两个长度为 n 的二进制字符串之间的汉明距离等于这两个字符串位不同的个数。写一个程序从命令行读入一个整数 k 和一个二进制字符串 s 并打印出所有和字符串 s 汉明距离不超过 k 的二进制字符串。举个例子，如果 $k=2$ 、 s 是 0000，那么你的程序需要打印出

0011 0101 0110 1001 1010 1100

提示：选择 s 中 k 位进行翻转。

- 2.3.22 递归正方形。写一个程序来生成下面的递归图案。这些正方形的大小之比为 2.2:1，要画一个有阴影的正方形，可以先画一个有填充的灰色正方形，再画一个未填充的黑色边框的正方形。



- 2.3.23 翻煎饼。在锅里有一摞大小不同的 n 个煎饼。你的目标是重新排列这一摞煎饼使得最大的煎饼在最下面，最小的煎饼在最上面。你仅仅可以翻转最上面的 k 个煎饼，从而反转它们的顺序。设计一个递归策略使得通过最多 $2n-3$ 次翻转就能正确排列这些煎饼。

- 2.3.24 格雷码。修改 Beckett (程序 2.3.3) 来打印格雷码 (不仅仅是那些改变了的位的序列)

295

- 2.3.25 汉诺塔变体。思考下面的汉诺塔问题的变体。三个柱子上共有 $2n$ 个尺寸递增的圆盘。初始状态所有奇数尺寸 (1, 3, ..., $2n-1$) 的圆盘放在左边的柱子上，从上到下按尺寸递增的顺序依次排列；所有偶数尺寸 (2, 4, ..., $2n$) 的圆盘按尺寸递增的顺序从上到下依次排列在右边的柱子上；写一个程序，使右边柱子上的圆盘移到左边柱子并把左边柱子上的圆盘移到右边。注意还需要遵守汉诺塔问题的规则。

- 2.3.26 动画版汉诺塔。使用 StdDraw 来把汉诺塔问题的解决过程用动画展示出来，大概一秒移动一个圆盘。

- 2.3.27 谢尔宾斯基三角形。写一个递归程序来绘制谢尔宾斯基三角形 (见程序 2.2.3)。像 Htree 一样使用命令行参数来控制递归深度。

- 2.3.28 二项分布。估计下面用来计算 binomial (100, 50) 的代码递归调用的次数：

```
public static double binomial(int n, int k)
{
    if ((n == 0) && (k == 0)) return 1.0;
    if ((n < 0) || (k < 0)) return 0.0;
    return (binomial(n-1, k) + binomial(n-1, k-1))/2.0;
}
```



谢尔宾斯基三角形

基于动态规划给出一个更好的解决方法。提示：见练习 1.4.41。

- 2.3.29 Collatz 函数。思考下面的递归函数。这个问题和数论中一个尚未解决的著名问题有关，这个问题被称为 Collatz 问题或 $3n+1$ 问题。

```
public static void collatz(int n)
{
    StdOut.print(n + " ");
    if (n == 1) return;
    if (n % 2 == 0) collatz(n / 2);
    else collatz(3*n + 1);
}
```

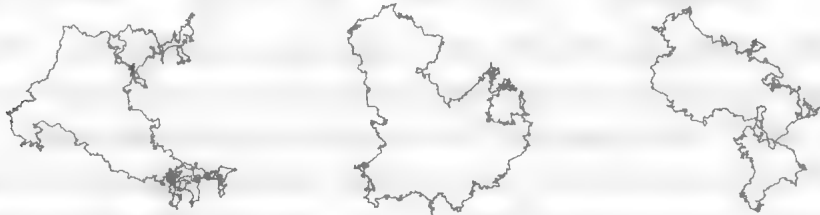
举个例子，一个对 collatz(7) 的调用会产生 17 次递归调用并打印如下序列：

296

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

写一个程序，以一个整数 n 作为命令行参数，返回一个整数 $i(i < n)$ ，在将所有小于 n 的整数作为 `collatz` 函数的参数调用时，`collatz(i)` 能够产生最大的递归调用次数。这个问题之所以未解决是因为没有人知道这个函数对任何整数是否都能在有限次内调用结束（数学归纳法无法解决这个问题因为总有一个递归调用会调用到更大的值）。

- 2.3.30 布朗岛。B.Mandelbrot 曾经提出过一个很著名的问题：英国的海岸线有多长？修改程序 `Brownian` 来获得一个新的可以画出布朗岛的程序 `BrownianIsland`（这个岛的海岸线看起来和大不列颠岛的海岸线很像）。这个修改很简单：第一，修改 `curve()` 函数，在 x 坐标和 y 坐标均添加随机高斯分布变量；第二，修改 `main()` 函数来从画布的中央开始画一条闭合的曲线。尝试在你的程序中使用不同的参数来画出看起来很逼真的岛屿。



Hurst 指数是 0.76 的布朗岛图

297

- 2.3.31 等离子体云。写一个可以画出等离子体云的递归程序，可以使用文中提到的方法。
- 2.3.32 一个奇怪的函数。思考如下麦肯锡 (McCarthy) 91 函数：

```
public static int mcCarthy(int n)
{
    if (n > 100) return n - 10;
    return mcCarthy(mcCarthy(n+11));
}
```

请不要使用计算机直接给出 `mcCarthy(50)` 的结果值。给出为了获得这个结果对 `mcCarthy()` 函数的调用次数。证明对所有正整数 n 都会收敛到基础步骤，或者给出 n 的值，使得递归进入无限循环。

- 2.3.33 递归树。写一个程序 `Tree` 使得它可以以一个整数 n 作为命令行参数，并且在 n 等于 1、2、3、4 和 8 时分别绘制出以下图形：



- 2.3.34 最长回文子序列。写一个程序 `LongestPalindromicSubsequence`，它以一个字符串作为命令行参数，并获得这个字符串的最长回文子序列（回文是指正着读和倒着读相同）。提示：计算这个字符串及其反转后的字符串的最长公共子序列。

298

2.4 案例研究：渗透

到目前为止，我们已经学过的编程工具足以让我们解决所有重要问题。本节我们将研究

并开发一个复杂程序来解决一个有趣的科学问题，并以此为案例来总结我们对方法和模块的学习。我们这样做的目的是结合具体问题，在解决各种挑战和困难的过程中，回顾我们所涵盖的基本要素，并展示一种可以广泛应用的编程风格。

在这个案例中，我们采用一种广泛适用的计算技术，称为蒙特卡罗模拟（Monte Carlo simulation），以研究一种称为渗透（percolation）的自然模型。“蒙特卡罗模拟”这一术语是指通过随机执行多次试验来评估未知量的计算技术，也被称为模拟（simulation）。前文中我们已经多次讨论并使用过这一技术，如赌徒破产问题和卡券收集问题。这一方法核心理念不是建立一个完整的数学模型或者测量一个实验的所有可能的结果，而是依靠概率法则体现模型的规律。

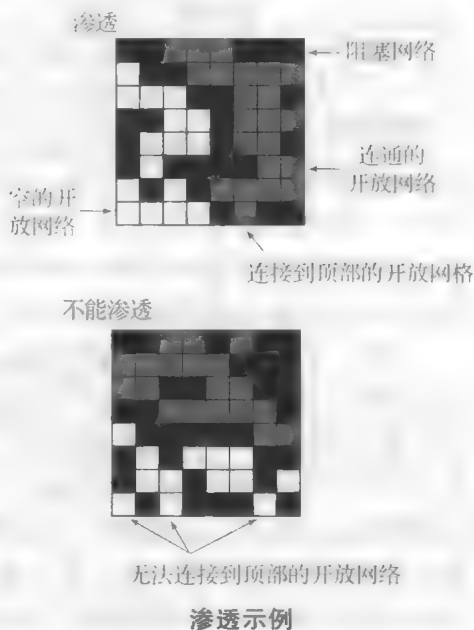
在本节的案例研究中，我们将学习一部分渗透相关的模型，这个模型是许多自然现象的基础。但是我们不会太多关注模型本身，我们的重点是开发模块化程序来解决计算任务的过程。我们找出可以独立处理的子任务，并努力确定关键的基础抽象概念，并自问如下问题：有没有哪些已往的子任务可以用来帮助解决这个问题？那些子任务的基本特征是什么？具有这些基本特征的子任务能不能帮助解决其他问题？提出这样的问题带来巨大的益处，因为它们引导我们开发更容易创建、调试和复用的软件，以便我们能够集中精力解决问题的关键部分。

渗透 在一些系统中，局部结构间的相互作用暗示着全局性质。例如，电气工程师可能对由随机分布的绝缘材料和金属材料组成的复合材料感兴趣：哪一部分材料需要是金属的，则整块材料可以导电？又如，地质学家可能对上表面有水（或下表面有石油）的多孔景观感兴趣。在哪些条件下，水能够渗透到底部（或石油可以喷涌到地表）？科学家已经定义了一个抽象过程来模拟这种情况，这一过程被称为渗透（Percolation）。它已被广泛研究并被证明是一种准确而有效的模型，它的应用多得令人眼花缭乱，包括绝缘材料、多孔物质、森林火灾的蔓延、疾病传播和互联网研究。

为了简单起见，我们从二维的渗透开始。我们将系统建模为 $n \times n$ 的网格，每个网格可能是阻塞的或者是开放的；开放网格的初始状态都是空的，如果一个开放网格可以通过与相邻位置（左、右、上、下四个方向）的开放网格的连接最终连到网格的最上一行，这个开放网格被称为连通网格（full site）。如果系统的底部行存在一个连通网格，那么我们说系统是渗透（percolates）的。换句话说，一个系统是渗透的就表示，如果我们在顶部行的开放网格注入物质，它会渗透到底部的开放网格。例如，对于绝缘/金属材料，开放网格对应金属材料，一个渗透系统即金属物质连通顶部到底部。对于多孔物质的例子来说，开放网格对应水可以流经的罅隙，一个渗透系统即水可以流经开放网格，从顶部到底部。

这是一个著名的科学问题，科学家们数十年来一直在深入研究。而其中的一个关键问题是：如果每个网格开放设置相互独立，互不影响，且网格开放的概率为 p （因此阻塞概率为 $1-p$ ），整个系统渗透的概率是多少？这个问题的数学解决方法尚未得出。我们的任务是编写计算机程序来帮助研究这个问题。

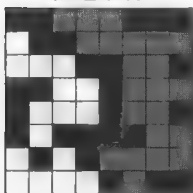
基础脚手架（scaffolding） 通过 Java 程序来解决渗透问题，我们面临着无数的决策和挑战，而且我们



还会编写大量的代码，比前文中我们已经学过的短程序要复杂得多。我们的目标是展示一种渐进式的编程风格，在这里我们独立地开发模块来解决问题的每一个部分，不断地通过设计和架构小型计算模块来增强信心，并最终解决整个复杂的问题。

第一步是选取数据的表现形式。这一决定可能会对我们以后编写的代码类型产生重大影响，因此不应掉以轻心。事实上，我们经常会在选择表现形式后不得不放弃它，并采用一个新的形式，这也是我们不断学习的过程。

渗透系统



阻塞网络

```
1 1 0 0 0 1 1 1
0 1 1 0 0 0 0 0
0 0 0 1 1 0 0 1
1 1 0 0 1 0 0 0
1 0 0 0 1 0 0 1
1 0 1 1 1 1 0 0
0 1 0 1 0 0 0 0
0 0 0 0 1 0 1 1
```

开放网络

```
0 0 1 1 1 0 0 0
1 0 0 1 1 1 1 1
1 1 1 0 0 1 1 0
0 0 1 1 0 1 1 1
0 1 1 1 0 1 1 0
0 1 0 0 0 0 1 1
1 0 1 0 1 1 1 1
1 1 1 1 0 1 0 0
```

连通网络

```
0 0 1 1 1 0 0 0
0 0 0 1 1 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 1
0 0 0 0 1 1 1 1
0 0 0 0 1 1 0 0
```

渗透表现

对于渗透问题，有效的数据表现形式是很显然的：使用一个 $n \times n$ 的数组。我们应该为每个元素使用哪种类型的数据？一种方案是使用整数，约定 0 表示一个开放网格，1 表示一个阻塞的网格，2 表示一个连通的网格。或者，请注意，我们描述网格是为了回答以下问题：网格是开放的还是阻塞的？网格是连通的还是空的？网格的这些特征表明，我们可以使用一个布尔类型的 $n \times n$ 数组来表示整个网格，其中每个元素可以用 `true` 或者 `false` 表示。我们把这样的二维数组称为布尔矩阵。使用布尔矩阵编写的代码更易理解。

布尔矩阵是一种基础的数学工具，应用非常广泛。但是，Java 中没有对布尔矩阵数据操作的原生支持。但是，我们可以使用 `StdArrayIO`（见程序 2.2.2）来实现这些矩阵的读写操作。这个过程告诉我们一个基础而常用的编程原则：多花些时间把自己的库做得通用而完善，在日后的编程工作中就会不断得到回报。

虽然最终我们将使用随机数据，但是我们还是希望能够通过读取和写入文件进行交互，因为随机输入的调试程序可能使得开发的过程非常低效。使用随机数据，每次运行程序时都会得到不同的输入；而修复了一个 Bug 之后，你想要看到的是与刚刚使用的相同的输入，以检查修复是否有效。因此，最好从我们理解的一些特定情况开始调试程序，并将输入数据保存在格式与 `StdArrayIO` 兼容的文件中（先是维度，然后是按行优先顺序排列的 0 和 1 值）。

当你开始解决一个新问题，而这个问题需要涉及多个源代码文件时，通常你需要创建一个新文件夹（目录）以将这些文件与你可能正在处理的其他文件隔离开来。例如，我们可能会创建一个名为 `percolation` 的文件夹来

存储此案例研究所需的所有文件。我们可以从实现和调试读写渗透系统的基本代码开始这次的工作，接着是创建测试文件、检查文件是否与代码兼容等，然后再考虑渗透部分的工作原理。这种类型的代码有时被称为脚手架（scaffolding），很容易实现，但要确保一开始就是稳固的，这样在处理主要问题时就不会分心。

完成了上面的工作，现在我们可以转向核心代码，测试一个布尔矩阵表示的系统是否是可渗透的。我们可以把这个任务看作模拟顶部被水淹没会发生什么（是否流到底部？）。我们的第一个设计决定是我们将要有一个 `flow()` 方法，该方法以布尔矩阵 `isOpen[][]` 作为参数（该矩阵描述哪些网格是开放的），并返回另一个布尔矩阵 `isFull[][]`，用于指定哪些网格已被填满（即转变为连通网络——译者注）。目前，我们完全不必担心如何实现这一方法；我们只是决定如何组织计算。很明显，我们希望客户代码能够使用 `percolates()` 方法来检查 `flow()` 返回的数组是否在底部有任何连通的网格。

Percolation (程序 2.4.1) 汇总了这些设计方案。到目前为止, 代码中没有执行任何有趣的计算, 当我们完成这些代码的运行和调试这些代码之后, 就可以开始考虑解决实际问题了。不执行计算的方法 (如 `flow()`) 有时称为存根函数 (stub)。有了存根函数, 我们就可以在我们所需要的上下文中测试和调试 `percolates()` 和 `main()`。我们将程序 2.4.1 这样的代码称为脚手架, 就像建筑工人在搭建建筑物时使用的脚手架一样, 这种代码提供了我们开发程序所需的支持。在编程工作的一开始就应该充分实现和调试这些代码 (即使代码还不完整, 我们也需要它们), 为构建代码解决当前的问题提供了良好的基础。通常, 我们将这个比喻推进一步——在代码实现工作完成后移除脚手架 (或者用更好的程序替换它)。

303

程序2.4.1 渗透程序的脚手架

```
public class Percolation
{
    public static boolean[][] flow(boolean[][] isOpen)
    {
        int n = isOpen.length;
        boolean[][] isFull = new boolean[n][n];
        // 计算矩阵isFull[] []的代码应该写在这里
        return isFull;
    }

    public static boolean percolates(boolean[][] isOpen)
    {
        boolean[][] isFull = flow(isOpen);
        int n = isOpen.length;
        for (int j = 0; j < n; j++)
            if (isFull[n-1][j]) return true;
        return false;
    }

    public static void main(String[] args)
    {
        boolean[][] isOpen = StdArrayIO.readBoolean2D();
        StdArrayIO.print(flow(isOpen));
        StdOut.println(percolates(isOpen));
    }
}
```

n	系统大小 (n×n)
isFull[] []	连通网格
isOpen[] []	开放网格

为了开始开发渗透程序, 我们实现并调试这个代码, 它处理与计算直接关联的任务。函数 `flow()` 返回一个布尔矩阵, 给出连通网格的位置 (应该在代码中的占位符位置, 目前还没实现)。辅助函数 `percolates()` 检查返回矩阵的底部行, 以决定系统是否渗透。测试用客户程序 `main()` 从标准输入中读取一个布尔矩阵, 并打印该矩阵的 `flow()` 和 `percolates()` 的调用结果。

```
% more testEZ.txt
5 5
0 1 1 0 1
0 0 1 1 1
1 1 0 1 1
1 0 0 0 1
0 1 1 1 1
```

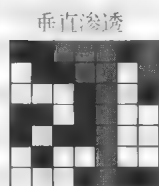
```
% java Percolation < testEZ.txt
5 5
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
false
```

304

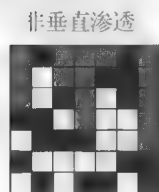
垂直渗透 给定一个表示开放网格的布尔矩阵, 我们如何确定它是否是一个渗透系统? 正如我们在本节后面会看到的, 这个计算结果直接关系到计算机科学的一个基本问题。目前, 我们将考虑一个更简单的问题, 我们称之为垂直渗透 (vertical percolation)。

解决这个问题的简化办法是只关注垂直连接路径。如果这样的路径从顶部到底部把系统连接了起来, 则我们说系统沿着路径垂直渗透 (并且系统本身垂直渗透)。如果我们谈论的是沙子穿过水泥, 而不是讨论水穿过水泥或电传导, 那么这个限制可能是直观而合理的。虽然

垂直渗透很简单，但它仍是一个有趣的问题，因为它衍生出了多个数学问题。这种限制与原来的系统有哪些显著的区别？我们期望有多少条垂直渗透路径？



通过垂直路径连接到顶部的网格



没有开放网格通过垂直路径连接到顶部
垂直渗透

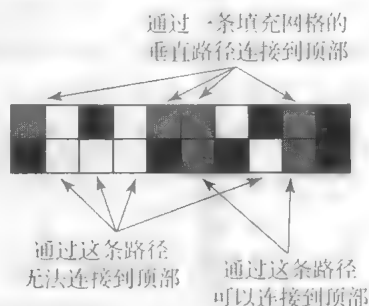
确定网格是否能够通过某个垂直路径连接到顶部是一个简单的计算过程。我们根据渗透系统的第一行初始化结果数组的第一行，然后从上到下，我们通过检查渗透系统矩阵的相应行来为结果数组的每一行元素赋值，即：在 `isopen[][]` 中有垂直连通上一行的连通网格，就在 `isFull[][]` 中将相应元素标记为 `true`。程序 2.4.2 是 Percolation 的 `flow()` 的一个实现，它返回整个网格的布尔矩阵（如果通过垂直路径连接到顶部，则为 `true`，否则为 `false`）。

测试 当我们确信我们的代码按照计划运行之后，我们希望在更广泛的测试用例上运行代码，并以此来解决一些科学问题。在这一点上，我们最初的脚手架变得不那么有用了，因为在标准输入和标准输出上用 0 和 1 表示大型的布尔矩阵，并进行大量的测试并不是一件明智的事。相反，我们希望自动生成测试用例，并观察代码的运行情况，确保它能够按照我们的预期运行。具体来说，为了保证代码的

305 正确性，并且更好地了解渗透，我们接下来的目标是：

- 使用大型随机布尔矩阵测试我们的代码。
- 依据给定的概率 p 评估系统渗透的概率。

为了实现这些目标，我们需要新的客户程序，这些客户程序要比用来启动和运行程序的脚手架稍微复杂一些。我们的模块化编程风格是在独立的类中开发这样的客户程序，无须修改我们的渗透代码。



垂直渗透计算示意图

程序2.4.2 垂直渗透检测

```
public static boolean[][] flow(boolean[][] isOpen)
{ // 计算垂直渗透的连通网格
  int n = isOpen.length;
  boolean[][] isFull = new boolean[n][n];
  for (int j = 0; j < n; j++)
    isFull[0][j] = isOpen[0][j];
  for (int i = 1; i < n; i++)
    for (int j = 0; j < n; j++)
      isFull[i][j] = isOpen[i][j] && isFull[i-1][j];
  return isFull;
}
```

n	系统大小 (n×n)
isFull[][]	连通网格
isOpen[][]	开放网格

将这种方法替换为程序2.4.1中的存根函数，可以解决仅垂直渗透问题，以及按照预期处理我们的测试用例（详见正文）。

```
% more test5.txt
5 5
0 1 1 0 1
0 0 1 1 1
1 1 0 1 1
1 0 0 0 1
0 1 1 1 1
```

```
% java Percolation < test5.txt
5 5
0 1 1 0 1
0 0 1 0 1
0 0 0 0 1
0 0 0 0 1
0 0 0 0 1
true
```


数据可视化。如果使用 StdDraw 进行输出，我们就可以处理更大的问题实例。以下 Percolation 类的静态方法允许我们将布尔矩阵的内容以可视化的形式呈现为一个 StdDraw 画布，画布被细分成很多小正方形，每个小正方形对应一个网格：

```
public static void show(boolean[][] a, boolean which)
{
    int n = a.length;
    StdDraw.setXscale(-1, n);
    StdDraw.setYscale(-1, n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (a[i][j] == which)
                StdDraw.filledSquare(j, n-i-1, 0.5);
}
```

第二个参数 which 指定我们要给哪些方块填充颜色，以对应 true 的元素或 false 的元素。这种方法虽然有点偏离了计算的目的，但是它能够帮助我们将大型问题可视化。函数 show() 通过用不同颜色填充网格来代表布尔矩阵中的阻塞和连通的元素，这样就完成了一个渗透过程的可视化图像。

蒙特卡罗模拟。我们希望我们的代码可以在任何布尔矩阵中都能正常工作。此外，在这个科学问题的研究中我们往往需要使用随机布尔矩阵。为此，我们为 Percolation 添加另一个静态方法：

```
public static boolean[][] random(int n, double p)
{
    boolean[][] a = new boolean[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            a[i][j] = StdRandom.bernoulli(p);
    return a;
}
```

该方法对于任何一个给定的 n 可以生成一个大小为 $n \times n$ 的随机布尔矩阵，每个元素为真的概率为 p 。

在几个特定的测试用例上调试过我们的代码后，我们准备在随机系统上测试它。这种情况仍然可能会发现一些新的错误，所以还是要注意检查结果的正确性。但是对代码片段进行测试还是有意义的，在对一段代码进行小规模测试之后，把它们集成进大系统时我们就会对代码更有信心。而且在消除了明显的错误之后，可以集中精力关注新错误。

[307]

开发完了这些工具，我们就可以很容易地在更大型的试验上测试我们的 Percolation 代码了。PercolationVisualizer（程序 2.4.3）仅包含一个 main() 方法，它从命令行提取 n 和 p ，并显示渗透计算的结果。

这种客户程序是非常典型的。我们最终的目标是准确估计渗透概率的值，通过大量的试验就可以估算出来，而这个简单客户程序的作用是使我们借助一些大型案例的求解过程来进一步了解这个问题（同时获得对我们代码能够正常运行的信心）。在进一步阅读之前，建议你从本书官网下载并运行此代码以研究渗透过程。当你运行中等大小的 n 阶（比如，50 到 100）程序 PercolationVisualizer 和各种 p 值时，你会乐于使用这个程序来回答一些关于渗透的问题。例如，显然，当 p 值很低时，系统不会渗透，当 p 值非常高时，系统总是渗透的；当 p 取中间值时，系统如何表现？随着 n 的增加系统如何变化？

估计概率 我们程序开发过程的下一步是编写代码来估计随机系统（大小为 n ，网格开放概率为 p ）渗透的概率，我们称之为渗透概率（percolation probability）。为了估计它的值，

我们只需要进行足够多次数的试验，这与投掷硬币的研究没有什么不同（见程序 2.2.6），唯一的区别就是这次我们生成一个随机系统并检查它是否渗透。

PercolationProbability（程序 2.4.4）将这个计算封装在一个方法 estimate() 中，该方法需要三个参数 n 、 p 和 trials： n 为系统的大小、 p 为网格开放概率、trials 为生成的随机系统的次数。利用这三个参数计算系统的渗透概率，最终返回这个概率值。

我们需要进行多少次试验才能获得准确的估计？这个问题是通过概率和统计学的基本方法来解决的，这些方法超出了本书的范围，但是我们可以通过多次计算过程的直观感受来回答这个问题。通过几次 PercolationProbability 的运行，你可以得知，如果网格开放概率接近于 0 或 1，那么我们不需要很多的试验，但是对于某些开放概率的值，我们需要多达 10000 次试验才能够将结果估计到小数点后两位。为了更详细地研究这种情况，我们可以修改程序，以达到类似 Bernoulli（程序 2.2.6）的输出，通过绘制数据点的直方图，以便看到值的分布（参见练习 2.4.9）。

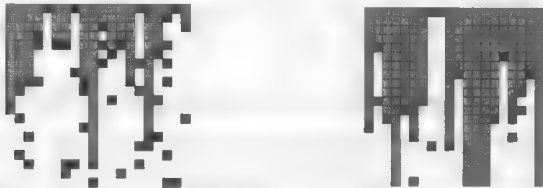
308

程序 2.4.3 可视化显示的客户端

<pre> public class PercolationVisualizer { public static void main(String[] args) { int n = Integer.parseInt(args[0]); double p = Double.parseDouble(args[1]); StdDraw.enableDoubleBuffering(); // 将阻塞网格用黑色填充 boolean[][] isOpen = Percolation.random(n, p); StdDraw.setPenColor(StdDraw.BLACK); Percolation.show(isOpen, false); // 将连通网格用蓝色填充（书中显示为灰色） StdDraw.setPenColor(StdDraw.BOOK_BLUE); boolean[][] isFull = Percolation.flow(isOpen); Percolation.show(isFull, true); StdDraw.show(); } } </pre>	<table border="1"> <tr> <td>n</td> <td>系统大小 ($n \times n$)</td> </tr> <tr> <td>p</td> <td>网格开放概率</td> </tr> <tr> <td>isOpen[][]</td> <td>开放网格</td> </tr> <tr> <td>isFull[][]</td> <td>连通网格</td> </tr> </table>	n	系统大小 ($n \times n$)	p	网格开放概率	isOpen[][]	开放网格	isFull[][]	连通网格
n	系统大小 ($n \times n$)								
p	网格开放概率								
isOpen[][]	开放网格								
isFull[][]	连通网格								

这个客户端使用两个命令行参数 n 和 p ，生成一个网格开放概率为 p 、大小为 $n \times n$ 的随机系统，并确定哪些网格是连通的，并在标准绘图中绘制。下图显示的是垂直渗透的结果。

% java PercolationVisualizer 20 0.9 % java PercolationVisualizer 20 0.95



309

使用 PercolationProbability.estimate() 会产生计算量的巨大飞跃，它会一下子产生数千万次的实验过程调用。如果没有彻底测试过我们的渗透方法，直接尝试这样做是很不明智的。另外，我们还需要注意完成计算所花费的时间。估计程序运行时间是 4.1 节的主题，目前这些程序的结构非常简单，我们可以很容易地估算它的运行时间，然后运行程序来验证我们的估算结果。如果我们进行 T 个试验，其中每个试验涉及 n^2 个网格，则

`PercolationProbability.estimate()` 的总运行时间与 n^2T 成正比。如果我们将 T 增加 10 倍（以获得更高的精确度），则运行时间将增加 10 倍。如果我们将 n 增加 10 倍（为了研究更大系统的渗透），运行时间会增加大约 100 倍。

对于一个具有几十亿个网格的系统，我们是否可以通过运行这个程序，将其渗透概率精确到小数点后多位？答案是否定的，因为没有计算机具备如此高的运行速度，能在可以接受的时间内执行完 `PercolationProbability.estimate()` 的调用。而且，在渗透的科学实验中， n 的值可能要高得多。我们可以假设能够在一个更大的计算机系统里进行这些试验，但任何计算机系统都不可能完全从原子量级上精确模拟真实世界的系统。因此，简化在科学中是至关重要的。

我们鼓励你从本书官网下载 `PercolationProbability`，以了解渗透概率以及计算它们所需的时间。在这个过程中，你不仅可以学习渗透程序的原理，而且还可以验证我们刚刚对其运行时间提出的假设。

一个网格开放概率为 p 的系统，其垂直渗透的概率是多少？垂直渗透相当简单，基本的概率模型可以为渗透概率产生精确的公式，并通过 `PercolationProbability` 来验证。由于我们研究垂直渗透的唯一原因是其可作为开发研究渗透方法的支撑软件的起点，所以我们将进一步研究一个垂直渗透系统作为练习题（参见练习 2.4.11），随后转向主要问题。

310

程序2.4.4 渗透概率估计

```
public class PercolationProbability
{
    public static double estimate(int n, double p, int trials)
    { // 生成随机大小n×n的试验系统，并返回
      // 渗透概率估计
      int count = 0;
      for (int t = 0; t < trials; t++)
      { // 生成一个随机大小n×n的布尔矩阵
        boolean[][] isOpen = Percolation.random(n, p);
        if (Percolation.percolates(isOpen)) count++;
      }
      return (double) count / trials;
    }
    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        double p = Double.parseDouble(args[1]);
        int trials = Integer.parseInt(args[2]);
        double q = estimate(n, p, trials);
        StdOut.println(q);
    }
}
```

n	系统大小 (n×n)
p	网格开放概率
trials	试验次数
isOpen[][]	开放网格
q	渗透概率

方法 `estimate()` 生成随机大小 $n \times n$ 的试验系统，其网格开放概率为 p ，并计算渗透概率。这是一个伯努利过程，像投掷硬币一样（见程序 2.2.6）。增加试验次数会增加估计的准确性。如果 p 接近于 0 或者 1，则不需要很多试验来实现准确的估计。程序运行结果如下。

```
% java PercolationProbability 20 0.05 10
0.0
% java PercolationProbability 20 0.95 10
1.0
% java PercolationProbability 20 0.85 10
0.7
% java PercolationProbability 20 0.85 1000
0.564
% java PercolationProbability 40 0.85 100
0.1
```

311

渗透的递归解决方案 在一般情况下，我们如何测试一个系统从顶部开始到底部的任一路径（而不仅仅是垂直的）是否渗透？

这个问题可以使用一个简洁的程序来解决，这一个经典的递归解决方案即深度优先搜索（depth-first search）。程序 2.4.5 中的方法 `flow()` 就是这个方法的一个实现，可以用于计算矩阵 `isFull[][]`。该方法需要 4 个参数，分别是存储每个网格是否开放的矩阵 `isOpen[][]`、当前矩阵 `isFull[][]`，以及用于指定当前网络位置的行索引 i 和列索引 j 。基础步骤是满足以下任何一个条件，就不做任何操作立刻返回（我们将这种调用称为空调用）：

- i 或 j 在数组边界之外。
- 网格被阻塞（`isOpen[i][j]` 为 `false`）。
- 我们已经将该网格标记为连通的（`isFull[i][j]` 为真）。

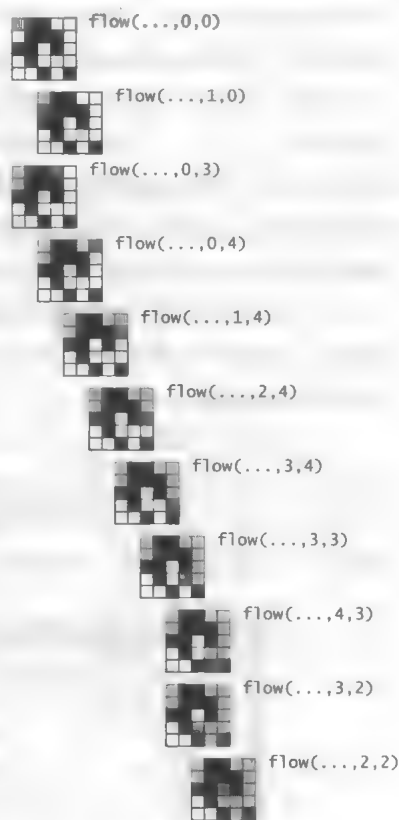
归约步骤是将网格标记为连通的并对它的四个邻居进行递归调用：`isOpen[i+1][j]`，`isOpen[i][j+1]`，`isOpen[i][j-1]`，`isOpen[i-1][j]`。单参数版本的 `flow()` 会为顶层行中每个网格调用这个递归版本的 `flow()` 方法。递归最终会停下来，因为每次递归调用要么是个空调用即什么也没做，要么会将一个网格标识为连通。我们可以通过数学归纳法来证明当且仅当一个网络与顶层是连通的，这个网格才会被标记为 `full`（就像其他递归程序的证明方法一样）。

在一个很小规模的测试用例中追踪 `flow()` 的执行过程是一个有益的练习。你会看到它为每一个可以通过开放网格的路径连接到顶层行的网格调用了 `flow()` 方法。从这个例子中可以看到，一些非常复杂的计算过程，可以通过递归程序以很简单的方式实现。该方法也是深度优先搜索算法的一个特例，具有很多重要的应用。

为了避免与我们的垂直渗透（程序 2.4.2）的解决方案发生冲突，我们可以将那个类重命名为 `PercolationVertical`，然后另外复制一份 `Percolation`（程序 2.4.1），用程序 2.4.5 中的两个 `flow()` 方法替换掉占位符 `flow()`。完成这些之后，我们可以用我们已经开发的 `PercolationVisualizer` 和 `PercolationProbability` 工具来执行这个算法的实验并以可视化的方式呈现结果。如果这样实施并尝试 n 和 p 的各种值，你会很快得到这样的直观感受：系统总是在网格开放概率 p 较高时渗透，而在 p 较低时不会渗透；特别是当 n 较大时，存在一个特殊的概率值，当 p 大于这个值时系统（几乎）总是渗透的，而低于这个值（几乎）就不会渗透。

因为已经在简单垂直渗透过程中对 `PercolationVisualizer` 和 `PercolationProbability` 进行了调试，我们可以更有信心地使用它们来研究渗透，并迅速开始研究我们感兴趣的科学问题。请注意，如果我们想再次尝试垂直渗透，那么我们需要修改 `PercolationVisualizer` 和 `PercolationProbability`，使它们引用 `PercolationVertical` 类而不是 `Percolation` 类，或者编写 `PercolationVertical` 和 `Percolation` 的新的客户程序来同时运行两个类中的方法来比较它们。

适应性绘图 为了更深入地了解渗透，程序开发的下一步是编写一个程序，将渗透概率作为给定的系统规模 n 和开放概率 p 的函数，绘制出函数曲线。也许产生这种绘图的最好方法是首先为该函数导出一个数学方程，然后用这个方程来绘制该图。然而，对于渗透问题，没有人能够推导出这样的方程，所以下一个选择是使用蒙特卡罗方法：运行模拟实验并绘制结果。



递归渗透（省略空调用）

[312]

程序2.4.5 渗透检测

```
public static boolean[][] flow(boolean[][] isOpen)
{ // 填充每一个能与顶行连接的网格
    int n = isOpen.length;
    boolean[][] isFull = new boolean[n][n];
    for (int j = 0; j < n; j++)
        flow(isOpen, isFull, 0, j);
    return isFull;
}

public static void flow(boolean[][] isOpen,
                        boolean[][] isFull, int i, int j)
{ // 填充每一个能与(i,j)连接的网格
    int n = isFull.length;
    if (i < 0 || i >= n) return;
    if (j < 0 || j >= n) return;
    if (!isOpen[i][j]) return;
    if (isFull[i][j]) return;
    isFull[i][j] = true;
    flow(isOpen, isFull, i+1, j); // 下
    flow(isOpen, isFull, i, j+1); // 上
    flow(isOpen, isFull, i, j-1); // 左
    flow(isOpen, isFull, i-1, j); // 右
}
```

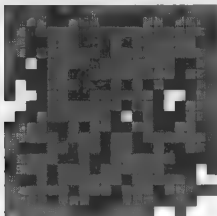
n	系统大小 (n×n)
isOpen[][]	开放网格
isFull[][]	连通网格
i, j	当前网格的行和列坐标

用这些方法替换程序2.4.1中的存根程序，可以得到渗透问题的基于深度优先搜索的解决方案。如果矩阵中的一个点可以通过一系列邻接的开放网格连接到isOpen[i][j]的点，那么递归函数flow()会将isFull[i][j]中对应的元素设置为true。单参数的flow()为顶行的每一个网格调用递归版本的flow()方法。

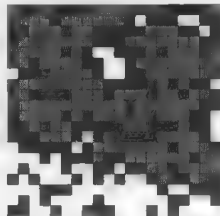
```
% more test8.txt
8 8
0 0 1 1 1 0 0 0
1 0 0 1 1 1 1 1
1 1 1 0 0 1 1 0
0 0 1 1 0 1 1 1
0 1 1 1 0 1 1 0
0 1 0 0 0 0 1 1
1 0 1 0 1 1 1 1
1 1 1 1 0 1 0 0
```

```
% java Percolation < test8.txt
8 8
0 0 1 1 1 0 0 0
0 0 0 1 1 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 1
0 0 0 0 1 1 1 1
0 0 0 0 0 1 0 0
true
```

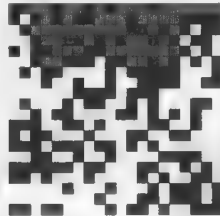
$p \approx 0.65$



$p \approx 0.60$



$p \approx 0.55$

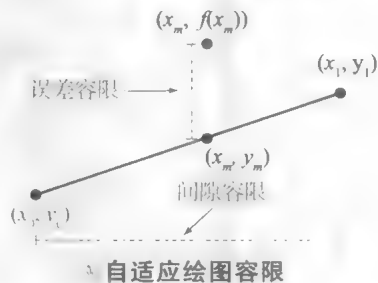


随着网格开放概率 p 降低，渗透的可能性降低

接下来，我们需要做出无数个决定。为了估计渗透概率，我们应该计算多少个开放概率 p ？我们应该为 p 选择哪些值？在这个模拟实验的过程中，我们的计算精度应该是多少？这些问题其实是在探讨我们应该如何合理地设计一个实验。我们很想为任何给定的 n 绘制一条准确的曲线，然而这样的计算成本可能是令人望而却步的。例如，首先你想到的是使用 StdStats (程序 2.2.5) 生成 100 或者 1000 个等距点，为每个点获得函数值就可以完成曲线绘制。但是，从程序 PercolationProbability 的实验中你已经得知，计算每个点的渗透概率的精确值可能需要几秒或更长时间，因此整个处理完所有这些点可能需要几分钟或几小时甚至更

长的时间。而且,显然很多计算时间是完全浪费的,因为我们知道小 p 的值是0,大 p 的值是1。我们可能更喜欢把时间花在精确地计算中间 p 值的过程中。我们应该如何着手?

PercolationPlot(程序2.4.6)实现了一个递归的方法,其结构与Brownian(程序2.3.5)相同,广泛适用于类似的问题。其基本思想很简单:我们设定 x 坐标值之间的最大距离(我们称之为间隙容限), y 坐标中允许的最大已知误差(它我们称之为误差容限),以及每个点的重试次数 T 。递归方法在给定区间 $[x_0, x_1]$ 内,在 (x_0, y_0) 到 (x_1, y_1) 之间绘图。对于我们的问题,图像是从 $(0, 0)$ 到 $(1, 1)$ 。基础步骤是如果 x_0 和 x_1 之间的距离小于间隙容限,或者连接两个端点的线与中点上的函数值之间的距离小于误差容限,则只须简单地画一条连接 (x_0, y_0) 到 (x_1, y_1) 的直线。归约步骤是(递归地)分别绘制曲线的两个半部分 (x_0, y_0) 和 $(x_m, f(x_m))$ 和 $(x_m, f(x_m))$ 与 (x_1, y_1) 。



PercolationPlot中的代码相对简单,并以较低的成本产生了效果不错的曲线。我们可以用它来研究 n 取不同值时曲线的形状变化,或者选择更小的容限来使得曲线更加接近实际值。理论上,这种估算方法的偏差是可以精确数学表述推导出来的,但是在探索和实验时过度细化可能是不恰当的,因为我们的目标只是建立一个关于渗透的假设,然后通过科学实验来验证它。

315

程序2.4.6 自适应绘图客户程序

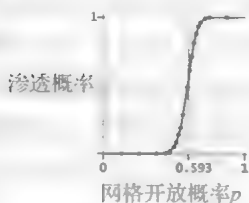
```
public class PercolationPlot
{
    public static void curve(int n,
                             double x0, double y0,
                             double x1, double y1)
    { // 进行实验并绘制曲线
        double gap = 0.01;
        double err = 0.0025;
        int trials = 10000;
        double xm = (x0 + x1)/2;
        double ym = (y0 + y1)/2;
        double fxm = PercolationProbability.estimate(n, xm, trials);
        if (x1 - x0 < gap || Math.abs(ym - fxm) < err)
        {
            StdDraw.line(x0, y0, x1, y1);
            return;
        }
        curve(n, x0, y0, xm, fxm);
        StdDraw.filledCircle(xm, fxm, 0.005);
        curve(n, xm, fxm, x1, y1);
    }

    public static void main(String[] args)
    { // 为n×n的渗透系统绘制结果曲线
        int n = Integer.parseInt(args[0]);
        curve(n, 0.0, 0.0, 1.0, 1.0);
    }
}
```

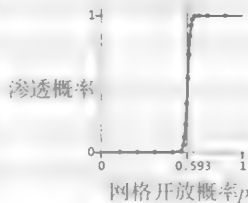
n	系统规模
x_0, y_0	左侧端点
x_1, y_1	右侧端点
x_m, y_m	中点
f_{xm}	中点处的值
gap	间隙容限
err	误差容限
trials	实验次数

这个递归程序绘制了在 $n \times n$ 的随机系统中,渗透概率(实验观测值)与网格开放概率 p (控制变量)的函数关系图。

% java PercolationPlot 20



% java PercolationPlot 100



316

事实上, PercolationPlot 产生的曲线立即证实了阈值存在的假设 (约 0.593): 如果 p 大于阈值, 那么系统几乎可以肯定地渗透; 如果 p 小于阈值, 那么系统几乎肯定不会渗透。随着 n 的增加, 曲线在阈值附近变化越剧烈, 最终接近一个阶跃函数, 在阈值处值从 0 改变为 1。这种现象称为相变 (phase transition), 在许多物理系统中都有发现。

程序 2.4.6 输出的简单形式掩盖了它背后的大量计算。例如, 对于 $n=100$ 绘制的曲线具有 18 个点, 每个是 10 000 次试验的结果, 每个试验涉及 n^2 个网格。生成和测试每个网格都需要执行若干行代码, 所以这个图是以执行数十亿条语句为代价的。从这个观察中可以学到两个教训。首先, 我们需要对可能执行数十亿次的任何代码行充满信心, 所以我们谨慎地逐步开发和调试代码是合理的。其次, 虽然我们可能对更大的系统感兴趣, 但我们需要进一步研究计算机科学来处理更大的情况, 也就是开发更快的算法以及有一套获得算法性能特征的框架。

如果我们复用以前实现过的所有代码, 只要实现不同版本的 flow() 方法, 就可以研究渗透问题的各种变体。例如, 如果在程序 2.4.5 的 flow() 方法中删掉最后一个递归调用, 它就变成了另外一种渗透模型。这种模型的特殊之处在于它不考虑上升的路径, 我们称之为定向渗透 (directed percolation)。这种模型可能对于通过多孔岩石的液体渗透 (其中重力作用使得液体不会向上渗透) 等类似的情况而言是重要的, 但是对于电连通性等类似的情况则不适用。如果你为两种方法都运行 PercolationPlot 方法, 你能辨别它们的差异吗 (见练习 2.4.10) ?

为了模拟诸如流经多孔物质的水的物理情况, 我们需要使用三维数组。在三维问题中, 是否有类似的阈值? 如果是这样, 它的值是什么? 深度优先搜索对于研究这个问题是有效的, 但是增加一个维度需要我们更加关注确定一个系统渗透的计算成本 (参见练习 2.4.18)。科学家也研究了更复杂的网格结构, 这些网格结构可能无法很好地用多维数组建模——我们将在 4.5 节中看到如何建模这些结构。

通过计算机实验研究渗透是非常有趣的, 因为没有人能够通过几个自然模型从数学上获得渗透的阈值。科学家知道这个值的唯一途径就是使用 Percolation 等程序进行模拟。科学家需要做实验来验证渗透模型是否准确地反映了自然界所观察到的现象, 是否需要进一步改进模型 (例如, 使用不同的网格结构) 等。渗透实验只是一

```
PercolationPlot.curve()
PercolationProbability.estimate()
Percolation.random()
    StdRandom.bernoulli()
        :  $n^2$ 次
    StdRandom.bernoulli()
return
Percolation.percolates()
    flow()
        return
return
// 每个点一次
Percolation.random()
    StdRandom.bernoulli()
        :  $n^2$ 次
    StdRandom.bernoulli()
return
Percolation.percolates()
    flow()
        return
return
return
//  $T$ 次
Percolation.random()
    StdRandom.bernoulli()
        :  $n^2$ 次
    StdRandom.bernoulli()
return
Percolation.percolates()
    flow()
        return
return
return
return
```

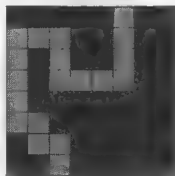
PercolationPlot 的函数调用跟踪

317

可以渗透 (没有向上的路径)



无法渗透



定向渗透

个例子，在这个问题中计算机科学成为整个科学研究过程的重要组成部分，随着你的学习，你会发现这样的问题会越来越多。

教训 如果我们试图设计和实现一个单一的程序来研究渗透问题，也许我们也能解决问题，也能够生成程序 2.4.6 绘制的类似曲线，但是写出的代码可能会多到数百行。在计算机技术发展的早期，程序员别无选择，只能编写和使用这样的程序，并花费大量的时间来找到 Bug 并纠正设计阶段的错误。而现在我们可以做得更好，我们可以使用像 Java 这样的现代编程工具，以及本章介绍的增量式编程风格的编程，而且需要牢记我们学到的一些经验教训。

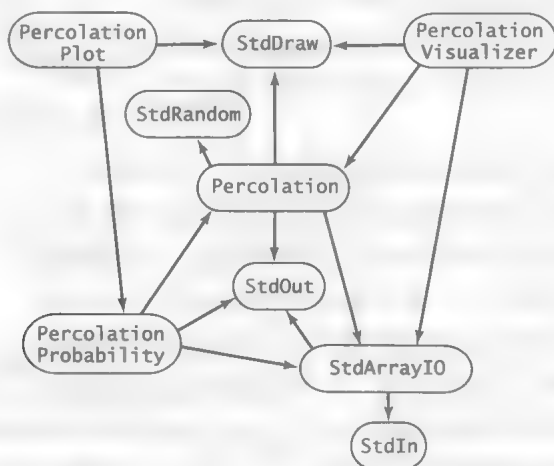
做好准备迎接 Bug。你写的每一段代码都非常有可能包含至少一个或两个错误，有时候甚至更多。编写一小段代码就开始调试运行，输入一些你能理解的小型测试数据，可以使你更轻松找到错误，并且更轻松修复它们。调试完成后，你可以把它们作为库函数，用于构建任何新的客户程序。

318

保持模块小巧。你一次只能关注最多几十行代码，因此你可以在编写代码时将代码分解为小模块。如果你在编程函数库，一些包含相关方法库的类最终可能会包含数百行代码；在其他情况下，我们应该使用小文件。

限制交互。在一个精心设计的模块化程序中，大多数模块应该仅仅依赖于数量有限的几个模块。具体来说，调用大量其他模块的模块需要被分成更小的块。被其他模块大量调用的模块（这样的情况在你的程序里应该不多）需要特别注意，因为如果你需要在模块的 API 中进行修改，则必须在所有客户程序中做出相应的调整。

逐步开发代码。对于每一个小模块，你应该开发完后就立刻运行和调试它们。这样在任何时候，你需要面对的不可靠的代码都不会超过几十行。如果你把所有小模块组合成一个大模块，很难确信其中任何一个都没有 Bug。尽早运行代码也会迫使你尽早考虑 I/O 格式、问题实例的属性以及其他问题。在考虑这些问题和调试相关代码时获得的经验使得你在后面的代码开发工作中更加高效。



案例研究依赖关系图（不包括系统调用）

解决一个更容易解决的问题。即使是能处理部分问题的解决方案，总比没有解决方案好，所以通常先把最简单的代码放在一起，这样就可以解决一个特定的问题，就像我们解决垂直渗透问题一样。这个解决方案会是我们在不断完善和改进的过程中的第一步。我

们可能通过测试更广泛的测试用例或者开发支持软件来更全面地理解问题，就像我们开发 `PercolationVisualizer` 和 `PercolationProbability` 类一样。

319

考虑一个递归的解决方案。递归是现代编程中不可或缺的工具，你应该学会并确信它能解决问题。如果你还没有被彻底说服，那么你可能希望尝试开发一个非递归程序来测试一个系统是否会渗透，拿它来与 `Percolation` 和 `PercolationPlot` 的简单和优雅相比较，并重新考虑这个问题。

适时地建立工具。我们的可视化方法 `show()` 和随机布尔矩阵生成方法 `random()` 对于许多其他应用程序当然是有用的，`PercolationPlot` 的自适应绘图方法也是如此。将这些方法合并到适当的库中将会很简单。相比于实现一些仅供渗透问题专用的方法，实现这些通用方法并不困难（有的时候可能更容易）。

尽可能复用软件。我们的 `StdIn`、`StdRandom` 和 `StdDraw` 库都简化了本节开发代码的过程，我们也可以立即复用诸如 `PercolationVisualizer`、`PercolationProbability` 和 `PercolationPlot` 等渗透程序来开发垂直渗透之后的程序。在你写了几个这样的程序之后，你可能会发现自己开发的这些程序可以在其他蒙特卡罗模拟的程序或其他实验数据分析问题中发挥作用。

这个案例研究的主要目的是要说服你，模块化编程是一种先进而有效的编程方法。虽然没有一种编程方法是万能的，但我们在本节中讨论的工具和方法将有助于你攻克复杂的编程任务，否则这些任务可能远远超出你的能力范围。

模块化编程的成功只是一个开始。相比于我们已经考虑过的类库即静态方法模型，现代编程系统具有更加灵活的编程模型。在接下来的两章中，我们将学习这个模型，并举例说明它的实用性。

320

问答环节

问：编辑 `PercolationVisualizer` 和 `PercolationProbability`，将 `Percolation` 重命名为 `PercolationVertical` 或任何我们想研究的方法似乎非常麻烦。有更简单的办法吗？

答：是的，这是第 3 章将要讨论的一个关键问题。除此之外，你也可以将不同的实现代码分别保存在不同的子目录中，但这可能会引起混淆。一些高级的 Java 机制（比如 `classpath`）也能解决这个问题，但是它们也有自己的问题。

问：递归版本的 `flow()` 方法让我感到头疼。我怎样才能更好地理解它在做什么？

答：生成一些你已经知道结果的数据示例，用来当作参数运行 `flow()` 方法，并在方法中加入打印函数调用跟踪的指令。经过几次运行后，你将获得足够的信心。它的作用就是，如果一个开放网格能够通过一系列开放网格的邻接关系连接到起始网格，就把这个网格标记为连通的。

问：是否有简单的非递归方法来识别连通的网格？

答：有几种方法可以执行基本相同的计算。我们将在 4.5 节中重新讨论这个问题，到时我们会学习一种新的算法：广度优先搜索（`breadth-first search`）。如果你感兴趣的话，开发 `flow()` 的非递归实现是一个艰难的任务，也肯定是一个有益的练习。

问：`PercolationPlot`（程序 2.4.6）似乎用了大量的计算才产生了一个简单的函数图。有更好的方法吗？

答：最好的解决方案当然是找到描述这个函数的一个简单的数学公式，但是几十年来科学家们无法得出这个公式。在科学家找到这样一个公式之前，他们必须求助于本节中的计算实验。

321

练习

2.4.1 编写一个程序，它需要命令行参数 n ，并创建一个 $n \times n$ 的布尔矩阵，假设一个元素的行列坐标为 (i, j) ，如果 i 和 j 互质，则设置该元素为 `true`，否则为 `false`。在标准绘图上显示该矩阵（参见练习 1.4.16）。然后，编写一个类似的程序来绘制 n 阶 Hadamard 矩阵（参见练习 1.4.29）。最后，编写一个程序来绘制一个布尔矩阵，对于矩阵中坐标为 (i, j) 的元素，如果 $(1+x)^j$ 的展开二项式中的项 x^i 的系数是奇数（见练习 1.4.41），那么该元素被设置为 `true`，否则为 `false`。你可能会对第三个例子形成的图形感到惊讶。

2.4.2 为 Percolation 实现一个 `print()` 方法，用“1”表示阻塞的网格，“0”表示开放的网格，“*”表示连通的网格。

2.4.3 给定如下输入，写出程序 2.4.5 中 `flow()` 的递归调用过程：

```
3 3
1 0 1
0 0 0
1 1 0
```

2.4.4 编写一个 Percolation 的客户程序，这个客户程序命令行读取系统模块 n 和增量 d ，对于 $n \times n$ 的随机系统，网格开放概率 p 从 0 递增到 1，每次的增量为 d ，进行一系列实验并用类似 PercolationVisualizer 的方式展示结果。

2.4.5 在没有阻塞网格的系统上运行 Percolation 程序时，描述网格被标记为连通的顺序。哪个是最后一个标记的网格？递归的深度是多少？

2.4.6 使用 PercolationPlot 绘制各种数学函数（将 `PercolationProbability.estimate()` 替换为相应的数学函数的估算语句）的实验。尝试函数 $f(x) = \sin x + \cos 10x$ ，看看曲线与正弦曲线的匹配度如何。试着用这个方法绘制你自己选择的三四个函数曲线。

2.4.7 把 Percolation 程序修改成动画展示方式，逐渐显示网格被逐个填充的过程。也可以用动画展示上一题中函数逐渐拟合的过程。

2.4.8 修改 Percolation 程序以计算求解过程中出现的最大递归深度。绘制该值与网格开放概率 p 的函数关系图。如果反转递归调用的顺序，你的答案如何改变？

2.4.9 修改 PercolationProbability 以产生类似于 Bernoulli（程序 2.2.6）所产生的输出。附加：使用你的程序来验证数据服从高斯分布的假设。

2.4.10 创建一个程序 PercolationDirected 测试定向渗透（通过在程序 2.4.5 的递归版 `flow()` 方法中放弃最后一个递归调用，如文中所述），然后使用 PercolationPlot 绘制一个定向渗透概率作为网格开放概率 p 的函数曲线。

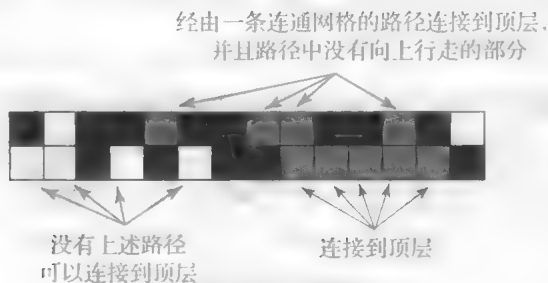
2.4.11 写一个 Percolation 和 PercolationDirected 的客户程序，从命令行中获取一个网格开放概率 p ，并打印出系统渗透但是不向下定向渗透的概率。使用足够的实验来保证得到的估计值精确到小数点后三位。

创新练习

2.4.12 垂直渗透。证明一个网格开放概率为 p 的系统垂直渗透概率为 $1 - (1 - p^n)^n$ ，并使用 PercolationProbability 程序验证不同的 n 值的情况。

2.4.13 矩形渗透系统。修改本节中的代码，以支持在矩形系统中研究渗透。比较宽高比为 2:1 和 1:2 的系统的渗透概率。

- 2.4.14 适应性绘图。修改 PercolationPlot，将其控制参数（间隙容限、误差容限和试验次数）作为命令行参数。试验各种参数值，以了解它们对曲线质量和计算成本的影响。简要描述你的发现。
- 2.4.15 非递归实现的定向渗透。编写一个非递归程序，在代码中通过从上到下移动来测试定向渗透。你的解决方案可以基于以下计算方法：如果在当前行中由开放网格组成的连续子行中，有任何网格连接到前一行上的某个连通网格，则这个子行中的所有网格都可以标识为连通的。



定向渗透的计算过程

- 2.4.16 快速渗透测试。修改程序 2.4.5 中的递归 flow() 方法，在找到底行的一个连通网格后立即返回（并且不再继续填充其他网格）。提示：使用一个参数，如果底部被击中，则返回 true，否则返回 false。在运行 PercolationPlot 时，粗略估计此更改带来的性能改进。使用需要至少运行几秒但不超过几分钟的 n 值。请注意，除非 flow() 中的第一个递归调用是针对当前网格正下方的网格，否则这种改进是无效的。

324

- 2.4.17 键渗透（Bond percolation）。假设网格的边缘之间是可以连通的，编写一个模块化的程序研究这种情况下的渗透。也就是说，边缘可以是空的也可以是连通的，并且如果存在由从上到下的连通边缘组成的路径，则系统渗透。注意：这个问题已经通过理论分析解决了，所以你的模拟实验应该验证分析的结论，即，当 n 足够大时，键渗透的阈值接近 $1/2$ 。

渗透



无法渗透



- 2.4.18 三维渗透。实现类 Percolation3D 和 BooleanMatrix3D（用于 I/O 和随机生成）来研究三维立方体的渗透，以推广本节研究的二维情况。渗透系统是由单位立方体组成的 $n \times n \times n$ 立方体，每个立方体以概率 p 开放并以概率 $1-p$ 阻塞。当两个开放的立方体有任何一个面相邻时（六个相邻面之一，除了边界上），渗透的路径都可以把它们连接起来。如果存在将底部平面上的任何开放的网格连接到顶部平面上的任何开放的网格的路径，则该系统渗透。使用递归版本的 flow()（如程序 2.4.5），但这次需要使用 8 个递归调用而不是 4 个。将渗透概率与网格开放概率 p 绘制成函数曲线，选择尽可能大的值 n 。如本节所强调的那样，务必逐步开发你的解决方案。

- 2.4.19 三角形网格上的键渗透。写一个模块化的程序，用于研究三角形网格上的键渗透，其中系统由 $2n^2$ 个等边三角形组合成 $n \times n$ 的菱形网格。每个内部点有六条边；边上的每个点有四条边；每个角点有两条边。

渗透



无法渗透



- 2.4.20 生命的游戏。编写一个类 GameOfLife 用于模拟康威的生命游戏（Conway's Game of Life）。我们用一个布尔矩阵来模拟这个系统，矩阵中的每一个元素对应于系统的一个单元，它的状态可能是活的或者是死的。游戏的过程中需要检查和更新每个单元格的值。每个单元格的值取决于它的邻居（每个方向上的相邻单元格以及对角线上的单元格）的值。在整个游戏的过程中，单元格的状态只能按照

325

以下条件发生变化，其他状况下均保持原状态不变：

- 如果一个死的单元格有且仅有三个活着的邻居，那么死的单元格会复活。
- 如果一个单元格有且仅有一个活着的邻居，那么这个单元格会死掉。
- 如果一个单元格有三个以上活着的邻居，那么这个单元格会死掉。

系统的初始状态是一个随机的布尔矩阵，或者使用本书官网上的任意一种模型作为初始状态，运行你的实验并查看最终结果。这个游戏已被深入研究，从原理层面涉及计算机科学的基础（参见本书官网获取更多信息）。



5 次迭代的过程

326
327

面向对象编程

学习高效编程的过程从理论上讲非常简单。到目前为止，我们已经学习了如何使用内置数据类型，接下来将在本章学习如何使用、创建和设计更高级的数据类型。

抽象是某种事物的简化描述，抓住事物本质的同时忽略所有其他细节。在科学、工程和编程领域，人们一直努力地通过抽象的方法来理解复杂的系统。在 Java 程序设计中，使用的是面向对象程序设计（object-oriented programming）方法，即将一个庞大而复杂的程序分解为一组交互的元素或对象。面向对象的程序设计的思想源于（在软件中）对真实世界的实体（如电子、人、建筑物或太阳系等）进行建模的方法，并且逐渐拓展到对二进制位、数字、颜色、图像或程序等抽象实体进行建模。

一个数据类型是一组值和定义在这些值上的一系列操作。在 Java 语言中，许多内置数据类型（如 int、float 类型）的取值范围和操作是预先定义好的。在面向对象的程序设计中，我们通过编写 Java 代码定义新的数据类型。一个对象是一个保存某种数据类型值的实体；你可以通过调用此数据类型的方法来操作此对象。

这种定义新的数据类型并处理保存数据类型值的对象的能力也称为数据抽象（data abstraction）。数据抽象引导我们采用模块化程序设计风格，这自然地拓展了面向过程程序设计风格，而后者则是第 2 章的基础。数据类型允许我们将数据及数据上的函数操作分离。本章我们遵循的程序设计理念是：在计算中应当清晰地分离数据和相关的计算任务。

328
329

3.1 使用数据类型

组织数据以备进一步处理是计算机程序开发的重要步骤。在使用 Java 语言进行程序设计时，这个过程主要是通过数据类型实现的。在 Java 中大量使用的是引用类型，旨在支持面向对象编程，这是一种便于组织和处理数据的编程风格。

在 Java 语言中，我们有 8 种基本数据类型（boolean、byte、char、double、float、int、long、和 short）可供使用，除此之外，还有大量的引用类型以及配套的库作为补充，这些引用类型为大量的应用程序量身定制。我们使用过的 String 数据类型就是这样一个例子。本节将学习有关 String 数据类型的更多信息，以及如何使用其他几种引用类型来进行图像处理和输入输出。这些引用类型中有些内置在 Java 中（String 和 Color），有些是基于本书（In、Out、Draw 和 Picture）开发的，这些后开发的数据类型作为通用资源具有重要的作用。

我们很容易注意到，本书前两章中的程序主要局限于数字操作。这是必然的，原因是 Java 的基本类型只能表示数字。String 类型是一个例外，这是一种内置在 Java 中的引用类型。使用引用类型后，我们编写的程序不仅可以处理字符串，还可以处理图像、声音或者 Java 库和本书网站上提供的数百种其他抽象数据类型。在本节中，我们重点关注使用现有数据类型的客户程序，从而为读者提供理解这些新概念的具体实例以作为参考，并阐述其应用

范围。我们将讨论处理字符串、颜色、图像、文件和 Web 页面的程序，本节涉及的内容与第 1 章基于基本类型的程序设计相比是一次大的飞跃。

在 3.2 节，我们将学习如何定义自己的数据类型以实现抽象，从而实现另一个飞跃，进入全新的编程世界。编写基于自定义数据类型的程序是一个非常强大和有用的程序设计风格，这种风格多年来一直主导着整个编程领域。

[330]

基本定义 数据类型是一组值的集合和定义在这些值上的一组操作的集合。这句话很重要，它是我们反复强调的几个观点之一。在第 1 章，我们详细讨论了 Java 内置数据类型。例如，基本数据类型 `int` 是 -2^{31} 到 $2^{31}-1$ 之间的整数；定义在 `int` 数据类型上的操作包含基本算术和比较操作，如 `+`、`*`、`%`、`<` 和 `>`。

我们也使用过非内置数据类型——`String` 类型。`String` 数据类型的值是字符序列，可以执行连接操作，这个操作需要输入两个字符串作为参数，并返回一个字符串当作结果。在本节中，我们将了解到用于处理字符串的其他几十个操作，如查找字符串长度、从字符串中提取单个字符以及比较两个字符串等。

每种数据类型都是一组值和定义在这些值上的一组操作的集合，但是当我们使用数据类型时，我们重点关注操作，而不是值。当编写程序使用 `int` 或 `double` 类型的值时，我们不关心这些值在内存中是如何存储的（我们从来没有详细说明这些细节），在编写引用类型的程序时也是如此，如 `String`、`Color` 或 `Picture`。换句话说，使用一个数据类型时无须理解其具体实现（这是本书将反复强调的另一个程序设计理念）。

String 数据类型。我们将在面向对象程序设计的上下文中重新学习 Java 的 `String` 数据类型。这样做有两个原因。首先，从第一个程序开始就一直使用 `String` 数据类型，对我们来说这是一个熟悉的例子。其次，字符串处理是很多计算应用程序的关键。字符串是编译和执行 Java 程序以及执行其他许多关键计算的核心。它们也是信息处理系统的基础，对于许多商业系统至关重要。日常生活中人们使用字符串撰写邮件、博客，进行聊天或准备发布的文档。字符串也是多个科学发展领域的重要组成部分，特别是分子生物学。

我们编写程序声明、创建和操作 `String` 类型的值。我们首先描述 `String` 类型的 API，它描述了编程中可用的若干操作。接下来，我们使用 Java 语言机制来声明变量，创建对象来保存数据类型的值并调用实例方法来应用这些数据类型的操作。这些功能与内置数据类型的操作有所不同，但是你也很容易注意到它们的相似之处。

[331]

API。Java 类提供了定义数据类型的功能。在一个类中，我们指定数据类型的值并实现数据类型上的操作。依旧遵循我们的程序设计理念，即使用一个数据类型时无须理解其具体实现，因此，通过在 API（应用程序编程接口）中列出数据实例可以使用的方法，从而指定客户程序可以发起的行为，如同在静态方法中所做的一样。API 的目的是给基于该数据类型编写的客户程序提供信息。

下表总结了 Java 中 `String` 数据类型的 API 中最经常使用的实例方法；而完整的 API 有 60 多个方法！以下几个方法使用整数作为字符串中字符的索引；与数组一样，这些索引的起始位置是从 0 开始的。

第一个方法名与类名相同，且没有返回值，这是在定义一个特殊方法，称为构造函数。其余的条目定义了实例方法，这些实例方法和我们早已使用过的静态方法一样接受参数和

返回值，但是它们不同于静态方法：实例方法实现了对数据类型的操作。例如，实例方法 `length()` 返回了字符串中的字符个数，而 `charAt()` 返回指定位置的字符。

public class String (Java String数据类型)	
String(String s)	创建一个与s值相同的字符串
int length()	字符串的个数
char charAt(int i)	索引下标为i的字符
String substring(int i, int j)	索引下标从i到j-1之间的字符串
boolean contains(String substring)	此字符串是否包含substring?
boolean startsWith(String pre)	此字符串是否以pre开头?
boolean endsWith(String post)	此字符串是否以post结尾?
int indexOf(String pattern)	第一次出现pattern的位置索引
int indexOf(String pattern, int i)	在索引位置i之后第一次出现的pattern的索引下标
String concat(String t)	此字符串后添加t
int compareTo(String t)	字符串比较
String toLowerCase()	此字符串的小写字母形式
String toUpperCase()	此字符串的大写字母形式
String replaceAll(String a, String b)	此字符串中的a用b来替换
String[] split(String delimiter)	字符串被delimiter分割后的子字符串组
boolean equals(Object t)	此字符串的值是否与t的值相同
int hashCode()	整数的散列码

有关其他许多可用方法请参考在线文档或本书官网

Java 的 String 数据类型的 API (部分内容)

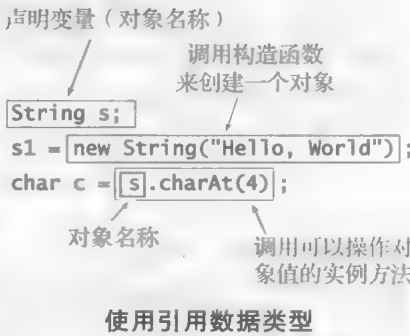
332

声明变量。声明引用数据类型变量的方式与声明内置数据类型变量的方式完全相同，声明语句都是数据类型名称后跟变量名称。例如下面这行语句：

```
String s;
```

作用是声明 `String` 类型的变量 `s`。此语句不会创建任何东西，仅仅是通知系统将会使用变量 `s` 来引用 `String` 类型的变量。为了编程风格统一，习惯上引用数据类型以大写字母开头，内置数据类型以小写字母开头。

创建对象。在 `Java` 中，每个数据类型的值都存储在一个对象中。要创建（或实例化）一个数据类型的对象（或实例），可调用构造函数指示 `Java` 创建一个对象。`Java` 调用构造函数时，使用关键字 `new` 后跟类名，并且指定构造函数的参数。构造函数的参数在括号中，并通过逗号分隔，这与静态方法是一样的。例如，`new String("Hello,World")` 创建一个新的 `String` 类型对象，该对象表示的字符序列是“`Hello,World`”。通常情况下，通过调用构造函数来创建一个新对象的同时，还在同一行代码中声明一个变量用来引用该对象：



```
String s = new String("Hello, World");
```

333 你可以创建任意数量的相同数据类型对象的对象，每个对象都应有其独立的标识，两个相同类型的对象存储的值可能相同也可能不同。例如，代码

```
String s1 = new String("Cat");
String s2 = new String("Dog");
String s3 = new String("Cat");
```

创建了三个不同的字符串对象。尽管 s1 和 s3 有相同的字符序列，但是它们并不是同一个对象。

调用实例方法。引用数据类型的变量和内置数据类型变量之间最大的区别是，使用引用数据类型变量可以调用为这个数据类型上的操作而专门定义的方法（与用于内置数据类型的操作（如 + 和 *）不同）。这类方法我们称为实例方法（instance method）。调用一个实例方法，与在另一个类中调用静态方法类似。只是实例方法不仅与类相关而且还与单独对象相关。因此，通常使用对象名（给定类型的变量）而不是类名来调用方法。例如，如果 s1 和 s2 是定义的 String 类型的变量，那么 s1.length() 返回整数 3、s1.charAt(1) 返回字符 'a'、s1.concat(s2) 返回一个新字符串 "CatDog"。

```
String a = new String("now is");
String b = new String("the time");
String c = new String(" the");
```

实例方法的调用	返回值类型	返回值
a.length()	int	6
a.charAt(4)	char	'i'
a.substring(2, 5)	String	"w i"
b.startsWith("the")	boolean	true
a.indexOf("is")	int	4
a.concat(c)	String	"now is the"
b.replace("t", "T")	String	"The Time"
a.split(" ")	String[]	{ "now", "is" }
b.equals(c)	boolean	false

String 数据类型的操作示例

String 类的快捷方法。Java 语言为 String 数据类型提供了专门的支持，有些你已经使用过，如可以使用字符串文本而无须调用构造函数来创建 String 对象。此外，也可以通过拼接操作符 (+) 而无须调用 concat() 方法来拼接两个字符串。在这里，单独介绍完整版方法的调用模式只是为了演示语法规则。这两个快捷方法只适用于 String 数据类型，你在其他数据类型里必须使用调用模式。

334	快捷版	String s = "abc";	String t = r + s;
	完整版	String s = new String("abc");	String t = r.concat(s);

以下代码片段说明了各种字符串处理方法的使用。这些代码清楚地展示了开发一个抽象模型的思想，并将抽象代码的实现与代码的使用相分离。这个能力是面向对象程序设计的特征，也是本书的一个转折点：虽然我们还没有看到按照这个规则实现的代码，但是从现在开始，本书编写的所有代码的风格都将是数据类型为中心，为了实现数据类型的某个操作而定义和调用方法。

从命令行参数中 提取文件名和扩展名	<pre>String s = args[0]; int dot = s.indexOf("."); String base = s.substring(0, dot); String extension = s.substring(dot + 1, s.length());</pre>
打印标准输入中包含 命令行参数的所有字符串	<pre>String query = args[0]; while (StdIn.hasNextLine()) { String line = StdIn.readLine(); if (line.contains(query)) StdOut.println(line); }</pre>
字符串是否为回文	<pre>public static boolean isPalindrome(String s) { int n = s.length(); for (int i = 0; i < n/2; i++) if (s.charAt(i) != s.charAt(n-1-i)) return false; return true; }</pre>
将 DNA 转译为 mRNA (用 U 来替换 T)	<pre>public static String translate(String dna) { dna = dna.toUpperCase(); String rna = dna.replaceAll("T", "U"); return rna; }</pre>

典型的字符串处理代码

335

字符串处理应用：基因组学 为了获得更多的字符串处理经验，我们将对基因组学领域做一个简单的介绍，并讨论一个程序，生物信息学家可以使用该程序发现潜在的基因。生物学家使用一个简单的模型来表示生命的构造：字母 A、C、T 和 G 分别代表生命体 DNA 中的四个碱基。在每个生命体中，这些基本构成部分组成一个很长的序列，称为基因组（每个染色体就是一个序列）。理解基因组的特性是理解其在生命体中自身表现过程的关键。许多已知生物的基因组序列（包括人类基因组），约由 30 亿个碱基构成。自从该序列被确定以后，科学家们已经开始编写计算机程序来研究它们的结构。字符串处理是现在分子生物学中最重要的方法之一，它使得研究过程具备实验性和可计算性。

基因预测。基因是基因组的一个子串，是理解生命过程至关重要的功能单元。基因由一系列的密码子组成，每个密码子是由一系列氨基酸组成，氨基酸是由三个碱基组成的序列。起始密码子 ATG 标记基因的开始，任何终止密码子 TAG、TAA 或 TGA 标记基因的终止（基因的其他位置不允许出现这些终止密码子）。分析基因组的第一步就是找到它的潜在基因，这是一个字符串的处理问题，可使用 Java 的 String 数据类型解决。

PotentialGene 程序（程序 3.1.1）是基因数据处理的第一步。isPotentialGene() 函数接收一个 DNA 序列作为参数，并根据以下标准确定它是否对应一个潜在基因：基因长度是 3 的倍数，以起始密码子开始，以终止密码子结束，并且在中间没有其他终止密码子。为了做出判断，程序使用了各种字符串的实例方法：length()、charAt()、startsWith()、endsWith()、substring() 和 equals()。

程序3.1.1 识别潜在基因

```

public class PotentialGene
{
    public static boolean isPotentialGene(String dna)
    {
        // 长度是3的倍数
        if (dna.length() % 3 != 0) return false;

        // 以起始密码子开始
        if (!dna.startsWith("ATG")) return false;

        // 没有终止密码子介入
        for (int i = 3; i < dna.length() - 3; i++)
        {
            if (i % 3 == 0)
            {
                String codon = dna.substring(i, i+3);
                if (codon.equals("TAA")) return false;
                if (codon.equals("TAG")) return false;
                if (codon.equals("TGA")) return false;
            }
        }

        // 以终止密码子结束
        if (dna.endsWith("TAA")) return true;
        if (dna.endsWith("TAG")) return true;
        if (dna.endsWith("TGA")) return true;

        return false;
    }
}

```

dna	待分析的字符串
codon	三个连续的碱基

isPotentialGene() 函数以DNA序列作为参数，并确定它是否对应一个潜在基因：长度是3的倍数，以起始密码子开始（ATG），以终止密码子结束（TAA、TAG或TGA），并且其中没有其他终止密码子。详细内容请见练习3.1.19的测试用客户程序。

```

% java PotentialGene ATGCGCTGCGTCTGTACTAG
true

% java PotentialGene ATGCGCTGCGTCTGTACTAG
false

```

尽管定义基因的规则比我们描述的要复杂一点，但是程序 PotentialGene 演示了程序设计基本知识是如何帮助科学家更有效地研究基因序列的。

在目前情况下，我们借助 String 数据类型展示的是定义数据类型的作用，即将一个重要的抽象进行良好的封装，并向客户程序提供有用的功能和方法。在继续讨论其他例子之前，我们讨论一下 Java 中引用类型和对象的一些基本属性。

对象引用。构造函数创建一个对象，并返回用户一个该对象的引用，而不是对象本身（因此命名为引用类型）。什么是对象的引用？这其实是一种访问对象的机制。Java 语言中有几种不同的方式来实现引用，但我们不需要知道具体细节。不过，应该在原理上知道一个通用的实现方法。一种容易理解的方法是，使用 new 分配内存空间来保存对象的数据值，并返回一个指向该空间的指针（内存地址）。此时，保存这个对象的内存地址也被称为对象的标识（identity）。

为什么不直接处理对象本身呢？对于一个小对象，这样做可能是可行的，而对大的对象来说，开销会成为问题：数据类型值可能消耗大量的内存。每次将对象作为参数传递给方法

时，复制或移动所有的数据是没有意义的。这个思路看起来很熟悉，我们之前在 2.1 节中将数组作为参数传递给静态方法时，就采用了相同的思考方式。事实上，数组就是对象，我们将在本节后面看到。相比之下，内置数据类型的值是直接在内存中存储的，所以不必使用引用类型的方式去访问每一个值。

在看过几个使用了引用类型的客户代码后，我们将更详细地讨论对象引用的属性。
使用对象。变量声明提供了一个对象的变量名称，我们可以在代码中使用变量名称，方法与使用 `int` 或者 `double` 类型的变量名一样。

- 作为方法的参数或返回值
- 在赋值语句中
- 在数组中

从 HelloWorld 程序以来，我们一直通过这种方式使用字符串对象：大多数程序使用字符串作为参数调用 `StdOut.println()`；所有程序都有一个 `main()` 方法，`main()` 的参数就是一个字符串类型数组。除了上面这些功能以外，引用对象的变量还有一个额外的重要功能：调用一个在引用类型上定义的实例方法。

此用法不适用于内置数据类型的变量，因为内置数据类型的操作是内置在语言中的，只能通过操作符（如 `+`、`-`、`*` 和 `/`）调用。

未初始化的变量。当声明一个引用类型的变量但并未赋值时，这个变量就处于未初始化（`uninitialized`）状态。这时，当你试图使用这个变量的时候，会产生与内置数据类型同样的问题。例如，

```
String bad;
boolean value = bad.startsWith("Hello");
```

代码会导致编译时错误：“`variable bad might not have been initialized`”（变量 `bad` 可能没有初始化），因为程序尝试使用一个未初始化的变量。

类型转换。如果要将对象从一种数据类型转换为另一种，必须编写代码来完成。通常情况下不存在这个问题，因为不同的数据类型具有不同的值，无法转换。试想，把一个 `String` 对象转换为一个 `Color` 对象有何意义呢？但有一个重要的场景下这种转换是值得的：Java 所有的引用类型都有一个特殊的实例方法 `toString()`，这个实例方法返回一个字符串对象。转换过程取决于代码的实现，程序员可以随意实现该方法，通常情况下会将对象的值进行编码并转换成一个字符串对象。通常，程序员在调试代码时调用 `toString()` 方法来打印跟踪。在某些情况下，Java 会自动调用 `toString()` 方法，如字符串连接和 `StdOut.println()`。也就是说，对于任何引用类型的对象 `x`，Java 都会自动将表达式 `"x"+x` 转换到 `"x="+x.toString()`，表达式 `StdOut.println(x)` 转换为 `StdOut.println(x.toString())`。在 3.3 节中，我们将研究实现此功能的 Java 语言内部机制。

访问引用数据类型。与静态方法的库一样，实现每个类的代码都保存在一个与该类同名



但带有 .java 扩展名的文件中。若要编写一个使用引用类型的客户程序，须使 Java 可以访问到该类对应的文件。String 数据类型是 Java 语言的一部分，因此它始终可用。通过将 .java 文件的副本放在与客户程序相同的目录中，或者使用 Java 类路径机制（在本书官网上有详细描述）可使用程序员自定义的数据类型。了解这个之后，我们将学习如何在客户代码中使用引用数据类型。

339

实例方法和静态方法的区别。最后，我们需要学习 static 修饰符的含义，自程序 1.1.1 以来我们一直在使用它——这是编写 Java 程序中最后神秘细节之一。静态方法的主要目的是实现函数；实例（非静态）方法的主要目的是实现数据类型的操作。我们的客户代码可以区分两种方法的使用，因为静态方法的调用通常以类名开头（按惯例大写），而实例方法调用通常以对象名称开头（按惯例小写）。这些差异总结在下表中，在编写了一些自己的代码之后，就能够快速识别差异了。

	实例方法	静态方法
调用样例	<code>s.startsWith("Hello")</code>	<code>Math.sqrt(2.0)</code>
调用	对象名（或对象引用）	类名
参数	调用方法的对象和参数	参数
主要的目的	操作对象的值	计算返回值

实例方法与静态方法

前文涉及的基本概念是面向对象编程的基础，所以这里非常有必要做一下总结。数据类型是一组值和定义在这组值上的一系列操作的集合。我们在独立模块中实现了数据类型，并在客户程序中使用它们。对象是数据类型的实例。一个对象的特点是它具有三个基本属性：状态、行为和标识。状态是对象当前表示的数据类型的值；行为由其数据类型的操作定义；对象的标识是其在内存中的存储位置。在面向对象程序设计中，我们通过调用构造函数创建对象，然后通过调用对象的实例方法修改它的状态。在 Java 中，我们通过对象引用来访问对象。

为了展示面向对象程序设计的强大能力，接下来我们讨论几个示例。首先，我们研究熟知的图像处理世界的问题，编程来处理 Color 和 Picture 对象。接着，我们从面向对象程序设计的角度回顾输入输出函数库，使我们能够从文件和 Web 页面中获取信息。

340

颜色 颜色是由电磁辐射引起的眼睛的感觉。因为在计算机中经常需要查看和处理彩色图像，所以颜色是计算机图形学中一个广泛应用的抽象，Java 为此提供了这样一种专门的数据类型即 Color。在专业出版、印刷、Web 等领域，处理颜色是一项复杂的任务。例如，彩色图像的显示效果在很大程度上取决于所使用的展示介质。Color 数据类型将创意设计所需颜色问题，转变为计算机系统忠实地呈现所需颜色的问题。

Java 库中有数百种数据类型，因此需要明确列出程序中使用了哪些 Java 库，以避免命名冲突。特别地，我们需要在使用 Color 的程序的开始处添加如下声明。

```
import java.awt.Color;
```

需要说明的是，到现在为止，我们一直在使用标准的 Java 库或我们自己编写的类，所以才一直没有使用库的导入功能。

Color 使用 RGB 颜色模型表示颜色值，即一种颜色由三个整数（每个取值范围为 0 到 255）确定，分别表示颜色的红、绿、蓝（相应）分量的强度。其他颜色值是通过混合红、

绿、蓝分量获得的。也就是说，Color的数据类型值是三个8位整数。我们不需要知道整数是使用int、short还是char值来表示的。根据这个模型，Java使用24位来表示每种颜色，可以表示 $256^3=2^{24} \approx 1670$ 万种可能的颜色。据科学家估计，人类肉眼可辨识的颜色只有约一千万种。

Color数据类型包含一个带3个整数参数的构造函数。例如，下面的代码创建了两种颜色，一种是纯红色，另一种是用来印刷本书英文原版封面的蓝色：

```
Color red      = new Color(255, 0, 0);
Color bookBlue = new Color( 9, 90, 166);
```

red	green	blue	
255	0	0	red
0	255	0	green
0	0	255	blue
0	0	0	black
100	100	100	dark gray
255	255	255	white
255	255	0	yellow
255	0	255	magenta
9	90	166	this color ⊖

一些颜色值

从1.5节开始，我们就开始使用模块StdDraw中的颜色，但是仅局限于若干预定义颜色，比如StdDraw.BLACK、StdDraw.RED和StdDraw.PINK。从现在开始，我们有数百万种颜色可供使用。AlbersSquares（程序3.1.2）是一个StdDraw客户端程序，可以用于各种与颜色有关的实验。

341

程序3.1.2 亚伯斯正方形

```
import java.awt.Color;

public class AlbersSquares
{
    public static void main(String[] args)
    {
        int r1 = Integer.parseInt(args[0]);
        int g1 = Integer.parseInt(args[1]);
        int b1 = Integer.parseInt(args[2]);
        Color c1 = new Color(r1, g1, b1);

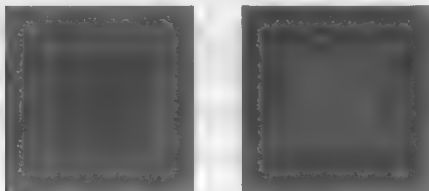
        int r2 = Integer.parseInt(args[3]);
        int g2 = Integer.parseInt(args[4]);
        int b2 = Integer.parseInt(args[5]);
        Color c2 = new Color(r2, g2, b2);

        StdDraw.setPenColor(c1);
        StdDraw.filledSquare(.25, 0.5, 0.2);
        StdDraw.setPenColor(c2);
        StdDraw.filledSquare(.25, 0.5, 0.1);
        StdDraw.setPenColor(c2);
        StdDraw.filledSquare(.75, 0.5, 0.2);
        StdDraw.setPenColor(c1);
        StdDraw.filledSquare(.75, 0.5, 0.1);
    }
}
```

r1, g1, b1	RGB值
c1	第一个颜色
r2, g2, b2	RGB值
c2	第二个颜色

这个程序从命令行接收两个RGB方式的参数值作为需要显示的颜色信息，并采用颜色理论家约瑟夫·亚伯斯（Josef Albers）在20世纪60年代开发的类似格式，约瑟夫·亚伯斯改变了人们思考色彩的方式。（彩色原图可通过华章网站（www.hzbook.com）下载和查阅）

% java AlbersSquares 9 90 166 100 100 100



342

像我们学习每一个新的抽象类时一样，我们只通过 Java 颜色模型的关键元素来介绍 Color，而不会深入所有的细节。Color 的 API 包含几个构造函数和 20 多个方法；常用的将在下面简单介绍。

```
public class java.awt.Color
```

```

    Color(int r, int g, int b)
    int  getRed()           红色的强度
    int  getGreen()         绿色的强度
    int  getBlue()          蓝色的强度
    Color brighter()        比此颜色更亮一些的颜色
    Color darker()          比此颜色更暗一些的颜色
    String toString()       能表示此颜色的字符串
    String equals(Object c) 颜色的值和对象c是否一样?
```

其他有用的方法可参考在线文档和本书网站

Java Color 数据类型的 API (部分)

我们的主要目的是用 Color 作为例子来说明面向对象程序设计，同时开发一些实用的工具，用于编写处理颜色的程序。接下来，我们选择一种颜色属性作为示例，以证明通过编写面向对象的代码来处理抽象概念（如颜色）是一种方便、有效的方法。




亮度。现代显示器（如 LCD 显示器、LED 电视和手机屏幕）上的图像质量依赖于一种颜色属性，称为单色亮度（monochrome luminance）或有效亮度。亮度的标准公式是根据眼睛对红色、绿色和蓝色的敏感度推导出来的。它是三种颜色分量的线性组合方程：如果一个颜色的红色、绿色和蓝色分量值分别是 r 、 g 和 b ，则其亮度 Y 的定义公式为：

$$Y=0.299r+0.587g+0.114b$$

由于系数为正且系数之和为 1，而各颜色分量的取值范围为 0 到 255 的整数，因此亮度的取值范围为 0 到 255 的实数。

灰度。RGB 颜色模型具有以下特性：当三种颜色分量强度相同时，所得结果颜色是位于黑色（全 0）到白色（全部 255）之间的灰度颜色。如果要在黑白报纸（或书籍）上印刷一幅彩色图像，则必须使用函数将其转换为灰度图像。将彩色图像转换为灰度图像最简单的方法是将红、绿、蓝分量值替换成其单色亮度值。




颜色兼容性。单色亮度也是确定两种颜色是否兼容的关键要素。两种颜色的兼容性是指在以一种颜色为背景时另一种颜色的可阅读性。一个广泛使用的经验法则是：前景色和背景色的亮度差至少应该是 128。例如，白色背景上的黑色文本具有 255 的亮度差，但黑色文本在本书英文版原书的蓝色背景下亮度差只有 74。这个法则在广告设计、道路标志、网站和许多其他应用程序的设计中十分重要。Luminance（程序 3.1.3）是一个静态方法的库，可以用于将一种颜色转换为灰度，也可以测试两种颜色是否兼容。程序中的静态方法说明了使用数据类型来组织信息的效用。因为如果不使用 Color，那么就会一次传递三个强度值，这个方法就变得非常麻烦，并且在没有引用类型的情况下，返回多个值是不可能的。使用 Color 对象作为参数和返回值大大简化了实现。

红	绿	蓝	此颜色 (英文版颜色)	
9	90	166		
74	74	74	灰度版本	
0	0	0	黑色	

$$0.299 * 9 + 0.587 * 90 + 0.114 * 166 = 74.445$$

灰度颜色示例（彩色原图可通过[华章网站 \(www.hzbook.com\)](http://www.hzbook.com) 下载和查阅）

343

亮度		差异
0		兼容 232
74		兼容 158
232		不兼容 74

颜色兼容性示例（彩色原图可通过华章网站（www.hzbook.com）下载和查阅）

程序3.13 亮度模块

```
import java.awt.Color;
public class Luminance
{
    public static double intensity(Color color)
    { // 颜色的单色亮度
      int r = color.getRed();
      int g = color.getGreen();
      int b = color.getBlue();
      return 0.299*r + 0.587*g + 0.114*b;
    }

    public static Color toGray(Color color)
    { // 使用亮度转换为灰度
      int y = (int) Math.round(intensity(color));
      Color gray = new Color(y, y, y);
      return gray;
    }

    public static boolean areCompatible(Color a, Color b)
    { // 如果颜色兼容则返回真，否则为假
      return Math.abs(intensity(a) - intensity(b)) >= 128.0;
    }

    public static void main(String[] args)
    { // 两种指定的RGB颜色是否兼容
      int[] a = new int[6];
      for (int i = 0; i < 6; i++)
        a[i] = Integer.parseInt(args[i]);
      Color c1 = new Color(a[0], a[1], a[2]);
      Color c2 = new Color(a[3], a[4], a[5]);
      StdOut.println(areCompatible(c1, c2));
    }
}
```

r, g, b | RGB值

y | color的亮度

a[] | args[]的整型值
c1 | 第一种颜色
c2 | 第二种颜色

该模块包含三个用于处理颜色的重要函数：单色亮度、将颜色转换为灰度、背景色和前景色兼容性测试。

```
% java Luminance 232 232 232 0 0 0
true
% java Luminance 9 90 166 232 232 232
true
% java Luminance 9 90 166 0 0 0
false
```

颜色抽象的重要性不仅仅在于可以直接使用，还可以用于构建包含 Color 值的高级数据类型。接下来，我们将通过开发一个建立在颜色抽象基础上的数据类型来说明这一点，该数据类型可用于编写处理数字图像的程序。

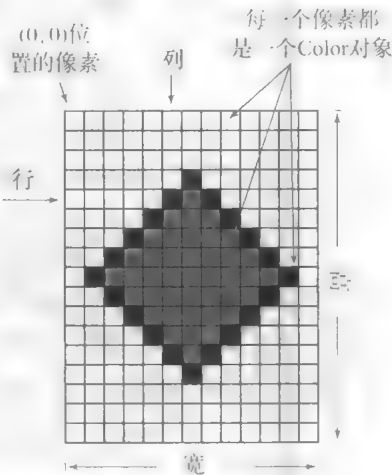
数字图像处理 你肯定熟悉照片的概念。从技术上讲，照片可以定义为通过对电磁波

辐射的可见光波长的收集和聚焦，构成一个场景在某个时间点的二维图像。这个技术定义超出了我们所研究的范围，值得注意的是摄影的历史是一个技术发展的历史。在 20 世纪，摄影基于化学过程，但是现代化摄影都是基于计算的。相机和手机其实就是一个带有镜头和感光器件、能以数字形式捕获图像的计算机，而且计算机具有照片编辑软件，可以处理这些图像。用户可以裁剪、放大和缩小图像，调整图像的对比度，增加或减少图像的亮度，去除红眼，或执行许多其他操作。给定一个简单的捕获数字图像思想的基本数据类型，许多诸如此类的操作则非常容易实现。

数字图像。处理数字图像需要哪些值？针对这些值需要执行哪些操作？计算机显示器的基本抽象和数字照片一致，同样非常简单：数字图像是一个像素（图片元素）组成的矩形网格，其中每个像素的颜色是单独定义的。数字图像有时被称为光栅或位图图像。与之对应的，使用 StdDraw 生成的图像类型（由点、线、圆和正方形等几何对象拼接而成）被称为矢量图像。

我们使用 Picture 数据类型实现数字图像的抽象。这组值是一个 Color 对象的二维矩阵，其操作包括创建一个给定宽度和高度的空白图像，从文件中加载一个图像，设置某个像素的颜色为给定的颜色，返回图像的宽度和高度，在计算机屏幕的窗口中显示图像，以及将图像保存为文件。在这个描述中，我们故意使用矩阵而不是数组来强调我们指的是一个抽象（一个像素矩阵），而不是一个特定的实现（一个 Java Color 对象的二维数组）。使用一个数据类型时无须理解其具体实现。的确，典型的图像具有如此多的像素，所以实现可能需要比 Color 对象的数组更有效的表示形式。在任何情况下，要编写处理图像的客户程序，只需要知道如下 API：

346



数字图像的解析

public class Picture		
Picture(String filename)		从文件创建一个图片
Picture(int w, int h)		创建一个w×h的空白图片
int width()		返回图片的宽度
int height()		返回图片的高度
Color get(int col, int row)		返回像素(col, row)的颜色
void set(int col, int row, Color c)		设置像素(col, row)的颜色为c
void show()		在窗口上展示图片
void save(String filename)		将图片保存为文件

我们的数据类型中用于图像处理的 API

按照惯例，(0, 0) 表示左上角的像素，所以图像的排列顺序与矩阵相同（相对地，StdDraw 模块的规则是点 (0,0) 位于左下角，以使得绘图的方向与笛卡儿坐标系的方式保持一致）。大多数图像处理程序其实就是过滤器，即先扫描源图像中的所有像素，然后执行一些计算以确定目标图像中每个像素的颜色值。第一个构造函数和 save() 方法支持 PNG 和 JPEG 格式，这两种格式使用非常广泛，因此你可以编写程序来处理自己的照片，并将处理

后的照片添加到相册或者上传到网站。show() 窗口提供一个交互式选项来保存文件。这些方法与 Java 的 Color 数据类型一起为我们打开了图像处理的大门。

灰度。本书网站中包括大量的彩色图像示例，本书描述的所有方法均适用于全彩图像，但是书中所有的示例图像都是灰度图。因此，我们的首要任务是编写一个程序，用于将彩色图像转换为灰度图像。这个任务是一个典型的图像处理基础任务：源图像中的每个像素对应于目标图像中一个不同颜色值的像素。Grayscale（程序 3.1.4）是一个过滤器，从命令行接收一个图像文件名称，并生成该图像的灰度版本。程序创建 Picture 对象，并初始化为彩色图像，然后将每个像素的颜色设置为一个新的 Color 值，新的 Color 值使用 Luminance（程序 3.1.3）中 toGray() 函数计算出的源图像像素点的灰度值。

347

程序3.1.4 将彩色图像转为灰度图像

```
import java.awt.Color;

public class Grayscale
{
    public static void main(String[] args)
    { // 以灰度显示图像
        Picture picture = new Picture(args[0]);
        for (int col = 0; col < picture.width(); col++)
        {
            for (int row = 0; row < picture.height(); row++)
            {
                Color color = picture.get(col, row);
                Color gray = Luminance.toGray(color);
                picture.set(col, row, gray);
            }
        }
        picture.show();
    }
}
```

picture	从文件中得到的图片
col, row	像素坐标
color	像素颜色值
gray	像素灰度值

这个程序是一个简单的图像处理客户程序。程序首先创建一个Picture对象，并用命令行参数命名的图像文件初始化该对象。然后通过计算每个像素的颜色的灰度值并用它来创建一个Color对象。用这个对象来设置对应像素的颜色，从而将图像中的每个像素转换为灰度。最后，显示转换后的图像。你可以在后文的图像中看见像素点，因为那些图像是从低分辨率图像放大而来（参见下一节图像缩放的内容）。

% java Grayscale mandrill.jpg



% java Grayscale darwin.jpg



348

缩放。图像处理最常见任务之一是放大或缩小图像。这种基本操作被称为缩放（scaling），它的应用很多，如制作用于聊天软件或手机上使用的缩略图照片、调整高分辨率照片的大小以适应印刷或 Web 页面的特定空间、放大卫星照片或使用显微镜拍摄的图像等。

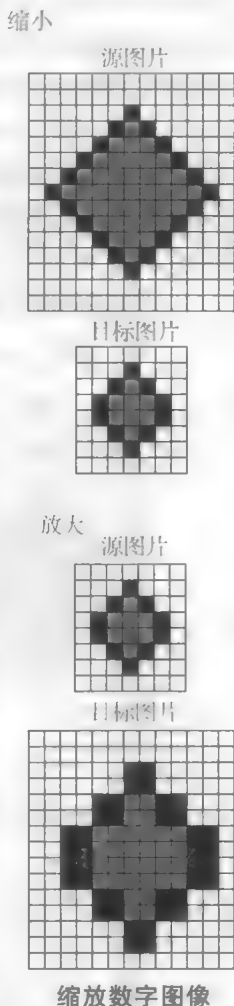
在光学系统中,我们仅仅通过调整镜头就可以达到所需的缩放大小,但在数字图像中必须要做更多的工作。

在某些情况下,缩放问题的实现策略变得非常简单清晰。例如,如果目标图像的大小是源图像的一半(每一个维度上都是一半),可以简单地通过选择一半像素来实现,即删除一半的行和列。这种技术被称为采样(sampling)。如果目标图像是源图像大小(每个维度)的两倍,可以设置目标图像中相邻的四个像素表示源图像中同一个像素的颜色。请注意,缩小图像会导致丢失信息,因此如果先把图像缩小一半再放大一倍,通常结果图像与源图像不一致。

一个简单的方案同样适用于图像的缩小和放大。我们的目标是生成一个缩放过的图像,所以从目标图像中的每个像素开始逐一处理,通过缩放每个像素的坐标以确定源图像中哪个像素的颜色可以用来赋值给目标图像的像素。假设源图像的宽度和高度分别用 w_s 和 h_s 表示,目标图像的宽度和高度分别用 w_t 和 h_t 表示,那么列索引坐标的缩放比例为 w_s/w_t ,行索引坐标的缩放比例为 h_s/h_t 。也就是说,目标图像第 c 列第 r 行的像素的颜色对应于源图像第 $c \times w_s/w_t$ 列第 $r \times h_s/h_t$ 行的像素颜色。例如,如果要将一幅图像的大小缩减到一半,则缩放比例为 2,因此目标图像的第 3 列第 2 行的像素颜色对应于源图像第 6 列第 4 行的像素颜色;如果将图像的大小变成两倍,则缩放比例为 $1/2$,因此目标图像的第 4 列第 6 行中的像素颜色由源图像的第 2 列第 3 行中的像素颜色获得。Scale(程序 3.1.5)是这个策略的一个实现。更复杂的方案可以有效地处理旧网页或旧相机中低分辨率的图像。例如,可以通过将源图像的每四个相邻像素点的平均值作为目标图像中的一个像素值。目前大多数应用中的图像通常为高分辨率图像,程序 Scale 采用的简单方法对处理高分辨率图像非常有效。

把目标图像的每个像素的颜色值视作源图像特定像素的函数,以计算像素的颜色值,这一基本思想同样是各种图像处理任务的有效方法。接下来,我们将讨论一个例子,更多的示例请参见练习和本书网站。

淡化效果。接下来我们讨论一个有趣的图像处理示例,即通过一系列离散的步骤,将一幅图像转换成另一幅图像。这种转变有时被称为淡化效果(fade effect)。Fade(程序 3.1.6)是 Picture 和 Color 的一个客户程序,程序使用线性插值法来实现这种效果。程序计算 $n-1$ 个中间图像,第 i 幅图像中的每个像素是源图像和目标图像中相应像素的加权平均值。静态方法 blend() 实现插值计算:源图像像素颜色的权重系数为 $1-i/n$,目标图像像素颜色的权重系数为 i/n (当 i 是 0 时,为源图像;当 i 是 n 时,为目标图像)。这个简单的计算方法可以产生惊人的结果。当在计算机上运行 Fade 程序时,变化将动态展现。尝试在你的照片库的一些图像上运行它。注意程序 Fade 假定图像具有相同的宽度和高度,如果两幅图像大小不同,则可以使用 Scale 为 Fade 创建一个或两个缩放的图像。

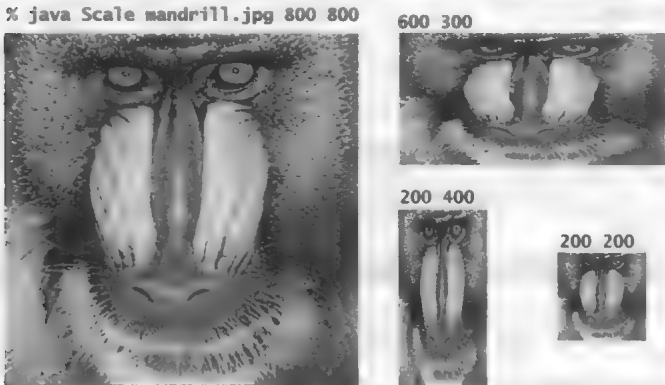


程序3.1.5 图像缩放

```
public class Scale
{
    public static void main(String[] args)
    {
        int w = Integer.parseInt(args[1]);
        int h = Integer.parseInt(args[2]);
        Picture source = new Picture(args[0]);
        Picture target = new Picture(w, h);
        for (int colT = 0; colT < w; colT++)
        {
            for (int rowT = 0; rowT < h; rowT++)
            {
                int colS = colT * source.width() / w;
                int rowS = rowT * source.height() / h;
                target.set(colT, rowT, source.get(colS, rowS));
            }
        }
        source.show();
        target.show();
    }
}
```

w, h	目标图像大小
source	源图像
target	目标图像
colT, rowT	目标图像像素坐标
colS, rowS	源图像像素坐标

程序接收三个命令行参数：图像的文件名称和两个整数（宽度w和高度h），将图像缩放为宽度为w、高度为h的大小，并显示缩放前后的图像。



350
}
351

回顾输入输出 在 1.5 节中，我们学习了如何使用 StdIn 和 StdOut 读写数字和文本，并使用 StdDraw 绘制图形。读者肯定已经体会到在程序中通过这些实用机制获取和输出信息的好处。使用这些模块的方便之处在于它们是“标准”接口，又是库函数，使得在程序的任何地方都可访问这些功能。但这些“标准”的一个缺点是访问文件依赖于操作系统的管道和重定向机制，并且任何给定程序只能访问一个输入文件、一个输出文件和一个绘制的图形。使用面向对象程序设计，可以定义与 StdIn、StdOut 和 StdDraw 类似的机制，而且允许在一个程序中同时使用多个输入流、输出流和图形。

我们将在本节中分别定义用于输入流、输出流和绘图的数据类型 In、Out 和 Draw。像往常一样，我们必须让 Java 可以访问到这些类（参见 1.5 节末尾的问答环节）。

程序3.1.6 淡化效果

```

import java.awt.Color;

public class Fade
{
    public static Color blend(Color c1, Color c2, double alpha)
    { // 计算颜色c1和c2的混合, 权重是alpha
        double r = (1-alpha)*c1.getRed() + alpha*c2.getRed();
        double g = (1-alpha)*c1.getGreen() + alpha*c2.getGreen();
        double b = (1-alpha)*c1.getBlue() + alpha*c2.getBlue();
        return new Color((int) r, (int) g, (int) b);
    }
    public static void main(String[] args)
    { // 显示从源图像到目标图像的m个渐变图像
        Picture source = new Picture(args[0]);
        Picture target = new Picture(args[1]);
        int n = Integer.parseInt(args[2]);
        int width = source.width();
        int height = source.height();
        Picture picture = new Picture(width, height);
        for (int i = 0; i <= n; i++)
        {
            for (int col = 0; col < width; col++)
            {
                for (int row = 0; row < height; row++)
                {
                    Color c1 = source.get(col, row);
                    Color c2 = target.get(col, row);
                    double alpha = (double) i / n;
                    Color color = blend(c1, c2, alpha);
                    picture.set(col, row, color);
                }
            }
            picture.show();
        }
    }
}

```

n	图像的数量
picture	当前图像
i	图像的计数器
c1	源图像像素颜色
c2	目标图像像素颜色
color	混合图像像素颜色

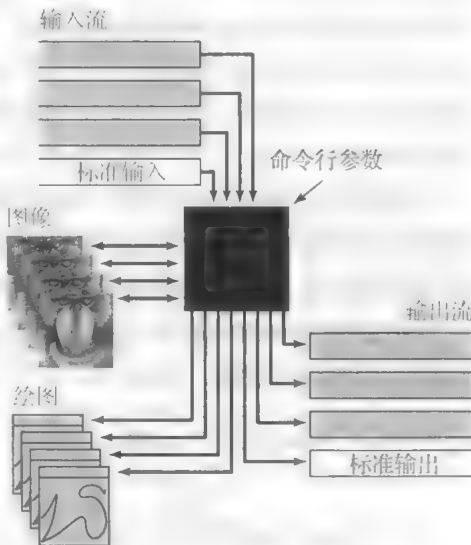
为了将一幅图像在 n 个步骤中淡化为另一张图像, 我们将第 i 个图像的每个像素设置对应源图像和目标图像的像素点的加权平均值, 其中源图像像素颜色的权重系数为 $1-i/n$, 目标图像像素颜色的权重系数为 i/n 。下列图片展示了这个例子中描述的转换过程。

% java Fade mandrill.jpg darwin.jpg 9



这些数据类型使我们能够灵活地处理 Java 程序中许多常见的数据处理任务。我们可以很容易地创建每种数据类型的多个对象，将数据流连接到不同的数据源和数据目标，而不是仅限于一个输入流、一个输出流和一个绘图。我们还可以灵活地设置变量来引用这些对象，将它们作为参数传递给函数或方法，或从函数或方法返回值，或者创建它们的数组，就像我们操作任何数据类型的对象一样操作它们。我们会在介绍完 API 后讨论几个应用实例。

输入流数据类型。我们的 In 数据类型是 StdIn 的一个更通用的版本，支持从文件、网站以及标准输入流中读取数字和文本。它实现了输入流 (input stream) 数据类型，其中 API 如下文所示。这种数据类型不仅仅局限于某一个抽象输入流 (标准输入)，还允许直接指定输入流的数据源。数据源甚至可以是文件或网站。当使用一个字符串参数调用 In 构造函数时，构造函数首先尝试在本地计算机中查找对应该文件名的文件。如果找不到该文件，则假设参数是一个网站名称，并尝试连接到该网站 (如果不存在这样的网站，则运行时会产生异常)。在这两种情况下，指定的文件或网站都将成为输入数据源，用于创建 In 对象，并且使用 read*() 方法从对应的流读取输入数据。



一个 Java 程序的鸟瞰图 (再次回顾)

public class In	
In()	从标准输入创建一个输入流
In(String name)	从文件和网页创建输入流
从输入流中读取特定元素的实例方法	
boolean isEmpty()	标准输入是否为空 (或只有空白格)
int readInt()	读取一个数据, 将其转换一个整型值, 然后将其返回
double readDouble()	读取一个数据, 将其转换一个双精度浮点值, 然后将其返回
...	
从输入流中读取字符的实例方法	
boolean hasNextChar()	标准输入是否有剩余字符?
char readChar()	从标准输入中读取一个字符并返回
从输入流中读取多行的实例方法	
boolean hasNextLine()	标准输入是否有下一行
String readLine()	读取该行的其余部分并将其作为字符串返回
读取输入流中其他部分的实例方法	
int[] readAllInts()	读取剩下的所有标记; 并作为整型数组返回
double[] readAllDoubles()	读取剩下的所有标记; 并作为浮点数组返回
...	

注意: In 对象也支持 StdIn 所支持的所有操作

352
353

这种设计使得在同一个程序中处理多个文件成为可能。此外，直接访问网络的能力可以将整个网络作为程序的潜在输入，如允许用户处理由其他人提供和维护的数据。这种类型的数据遍布整个 Web。科学家们现在定期上传测量结果或实验结果的数据文件，从基因组和蛋白质序列到卫星照片和天文观测；金融服务公司，如证券交易所也会定期发布关于股票和其他金融方面的详细信息；政府部分公布选举结果等。现在我们可以编写直接读取这些文件的 Java 程序。In 数据类型使得我们可以充分利用众多可用数据源，极大提高了程序的灵活性。

输出流数据类型。同样，Out 数据类型是 StdOut 的更通用的版本，它支持写入字符串到各种不同的输出流，包括标准输出和文件。同样，API 中指定了与 StdOut 相对应的方法。在调用构造函数时，指定一个文件名作为其参数，可以指定要用于输出的文件。Out 将参数字符串解释为本地计算机上的一个新文件名，并将结果写入该文件。如果调用 Out 构造函数时没有指定任何参数，结果将写入标准输出。

```
public class Out
```

Out()	创建一个输出到标准输出
Out(String name)	创建一个输出到指定的文件
void print(String s)	打印字符串s到输出流
void println(String s)	打印字符串s和换行符到输出流
void println()	打印换行符到输出流
void printf(String format, ...)	将参数打印到输出流上，并按照指定的format格式字符串打印

355

输出流数据类型的 API

程序 3.1.7 拼接文件

```
public class Cat
{
    public static void main(String[] args)
    {
        Out out = new Out(args[args.length-1]);
        for (int i = 0; i < args.length - 1; i++)
        {
            In in = new In(args[i]);
            String s = in.readAll();
            out.println(s);
        }
    }
}
```

out	输出流
i	参数索引
in	当前的输入流
s	当前文件的内容

该程序创建一个输出文件，其名称由程序最后一个命令行参数指定，输出文件的内容是一系列输入文件的拼接。一系列输入文件名由程序其他命令行参数指定。

```
% more in1.txt
This is
% more in2.txt
a tiny
test.
```

```
% java Cat in1.txt in2.txt out.txt
% more out.txt
This is
a tiny
test.
```

文件拼接和过滤。程序 3.1.7 是 In 和 Out 的客户端示例程序。程序使用多个输入流，将多个输入文件拼接成一个单独的输出文件。有些操作系统提供一个称为 cat 的命令行程序来实现这个功能。但是，实现类似功能 Java 程序也许更有用，因为可以不同方式修改 Java 程序来过滤输入文件，如忽略不相关的信息、更改格式、选择部分数据等。程序中讨论了其中一种处理的示例，其他的处理示例请参见练习。

356

Web 信息抓取。In 数据类型（用于通过 Web 网页创建一个输入流）和 String 数据类型（用于提供强大的工具来处理文本字符串）的结合，使得 Java 程序可以直接访问整个 Web，而无须依赖操作系统或浏览器。我们把这样的过程称为 Web 信息抓取：目标是用程序从网页中提取一些信息，而无须使用浏览器的浏览和搜索功能。要实现这个目标，可以充分利用许多网页为高度结构化的文本文件的特点（因为它们是由计算机程序创建的！）。浏览器具有允许用户查看正在浏览的网页的源代码功能，通过查看源代码可以猜测其结果。

假设我们以一个股票交易代码作为命令行参数，希望输出其当前的交易价格。股票信息由金融服务公司和互联网服务提供商在网上发布。例如，通过浏览网址 <http://finance.yahoo.com/q?s=goog>，可查看代号为 goog 的公司股票价格。同许多网页一样，这个网址中包含了一个参数（goog），我们可以将其替换为其他股票代码，以获得其他公司金融信息的网页。而且，就像网络上的许多其他文件一样，被引用的文件也是一个文本文件，使用一种称为 HTML 的格式语言编写。从 Java 程序的角度看，它可以看作通过一个

```
...
(GOOG)</h2> <span class="rtq
exch"><span class="rtq dash"> </span>
NMS </span><span class="wl sign">
</span></div></div>
<div class="yfi rt quote summary rt top
sigfig promo 1"><div>
<span class="time rtq ticker">
<span id="yfs_l84goog">1,100.62</span>
</span> <span class="down r time rtq
content"><span id="yfs c63 goog">
...
```

来源于 Web 的 HTML 代码

In 对象访问的 String 值。我们可以使用浏览器下载该文件的源代码，或者使用如下命令将源文件保存到计算机的本地文件 goog.html（尽管没有必要这么做）：

```
% java Cat "http://finance.yahoo.com/q?s=goog" goog.html
```

现在，假设 goog 的交易价格为 \$1,100.62。如果在源代码文件中搜索字符串“1,100.62”，你将发现股票价格出现在 HTML 代码中的某个位置。用户不必了解 HTML 的细节，只要弄清楚股票价格出现的上下文即可。在上述情况下，我们可以发现股票价格包含在子字符串 `` 以及 `` 之间。

357

使用 String 数据类型的 `indexOf()` 和 `substring()` 方法可以轻松获取股票价格信息，具体参见 StockQuote（程序 3.1.8）所示。该程序依赖于 <http://finance.yahoo.com> 使用的 Web 页面格式，如果其页面格式发生更改，StockQuote 可能无法正常工作。事实上，当读者阅读本书时，该 Web 页面格式可能已经改变了。即便如此，自己修改程序以适应变化的页面格式也不会很困难。读者可以通过各种有趣的方式来修改 StockQuote。例如，可以实现周期性获取股票价格并绘制股票图，计算股份变化的平均值，或将结果保存到一个文件以供后续分析使用。当然，同样的技术也适用于所有的 Web 数据源，更多信息请参见本节末尾的练习示例以及本书网站。

抽取数据。在程序中维护多个输入流和输出流，使得我们可以处理来自不同数据源的大量数据。我们将讨论一个示例：假设一个科学家或者一个金融分析师有保存在电子表格程

序中的海量数据。通常，电子表格是包含大量行和相对少的列的表格。用户感兴趣的可能并不是电子表格中的所有数据，而是少数几列。我们可以使用电子表格程序进行计算（毕竟使用电子表格的目的就是用于计算），但是电子表格肯定没有 Java 程序灵活。针对这种情况的解决方法是把电子表格的数据导出到文本文件中，并使用特殊字符来分隔列，然后编写一个 Java 程序从输入流中读取该文本文件。一种标准的最佳解决方案是使用逗号作为分隔符：每行数据占一行，每行的列数据用逗号分隔。这样的文件被称为逗号分隔文件或 .csv 文件。使用 Java 的 String 数据类型中的 split() 方法，可以实现逐行读取数据并分离各数据项。本书后面将会讨论这种方法的几个例子。Split（程序 3.1.9）是 In 和 Out 的客户程序，进一步实现了如下功能：程序创建多个输出流，每个文件对应一系列数据。

这些例子清晰地说明了使用多个输入输出流，操作文本文件是多么的便捷，这种机制甚至可以用于直接访问 Web 页面。Web 页面基于 HTML 编写，所以可以被任何可读取字符串的程序访问。之所以采用这些非常简单的文本格式，就像 .csv 这样，而不是依赖于具体程序的复杂数据格式，是为了允许尽可能多的人可以使用简单的程序（如 Split）访问数据。

358

程序 3.1.8 股票报价的Web信息抓取

```
public class StockQuote
{
    private static String readHTML(String symbol)
    { // 返回与股票代码对应的HTML
      In page = new In("http://finance.yahoo.com/q?s=" + symbol);
      return page.readAll();
    }

    public static double priceOf(String symbol)
    { // 返回当前股票代码的价格
      String html = readHTML(symbol);
      int p      = html.indexOf("yfs_184", 0);
      int from   = html.indexOf(">", p);
      int to     = html.indexOf("</span>", from);
      String price = html.substring(from + 1, to);
      return Double.parseDouble(price.replaceAll(",", ""));
    }

    public static void main(String[] args)
    { // 打印指定股票代码的价格
      String symbol = args[0];
      double price = priceOf(symbol);
      StdOut.println(price);
    }
}
```

symbol	股票代码
page	输入流
html	页面的内容
p	yfs_184索引
from	>索引
to	索引
price	当前的价格

该程序接受一个股票代码作为命令行参数，并在标准输出中写入当前股票的价格，当前股票价格来自于网站<http://finance.yahoo.com>上提供的实时信息。程序使用String中的indexOf()、substring()和replaceAll()方法。

```
% java StockQuote goog
1100.62
% java StockQuote adbe
70.51
```

359

程序 3.1.9 分割一个文件

```

public class Split
{
    public static void main(String[] args)
    { // 按列将一个文件分割成n个文件
        String name = args[0];
        int n = Integer.parseInt(args[1]);
        String delimiter = ",";

        // 创建输出流
        Out[] out = new Out[n];
        for (int i = 0; i < n; i++)
            out[i] = new Out(name + i + ".txt");

        In in = new In(name + ".csv");
        while (in.hasNextLine())
        { // 读取一行并将字段写到输出流
            String line = in.readLine();
            String[] fields = line.split(delimiter);
            for (int i = 0; i < n; i++)
                out[i].println(fields[i]);
        }
    }
}

```

name	基本文件名称
n	字段的数量
delimiter	分隔符 (逗号)
in	输入流
out[]	输出流
line	当前行
fields[]	当前行中的值

该程序使用多个输出流将.csv文件拆分为几个单独的文件。每个文件对应以逗号分隔的字段。与第i个字段相对应的输出文件的名称是通过将i和.csv连接到原始文件名的末尾形成的

```
% more DJIA.csv
```

```

...
31-Oct-29,264.97,7150000,273.51
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
24-Oct-29,305.85,12900000,299.47
23-Oct-29,326.51,6370000,305.85
22-Oct-29,322.03,4130000,326.51
21-Oct-29,323.87,6090000,320.91
...

```

```
% java Split DJIA 4
% more DJIA2.txt
```

```

...
7150000
10730000
16410000
9210000
5920000
12900000
6370000
4130000
6090000
...

```

360

绘图数据类型。本节前面讨论过 Picture 数据类型，使用这种数据类型，我们可以编写处理多个图片、图片数组等的程序，这正是因为这种数据类型提供了使用 Picture 对象进行计算的能力。当然，我们也希望拥有与 StdDraw 类似的创建几何对象的计算能力。因此，我们有一个具有以下 API 的 Draw 数据类型：

```
public class Draw
```

```
    Draw()
```

绘图命令

```

void line(double x0, double y0, double x1, double y1)
void point(double x, double y)
void circle(double x, double y, double radius)
void filledCircle(double x, double y, double radius)
...

```

控制命令

```

void setXscale(double x0, double x1)
void setYscale(double y0, double y1)
void setPenRadius(double radius)
...

```

注意：Draw数据类型对象也支持StdDraw所支持的所有操作。

对于任何数据类型，我们可以使用来 new 创建一个新的 Draw 对象并将其引用给一个变量，并使用该变量名称来调用创建图形的函数。例如，代码

```
Draw draw = new Draw();
draw.circle(0.5, 0.5, 0.2);
```

[361] 在屏幕上一个窗口的中心画一个圆圈。与 Picture 一样，每个图形都有自己的窗口，因此可以解决调用程序来同时展示多个图形的问题。

引用类型的属性。现在你已经见过几个引用类型的例子了（Charge、Color、Picture、String、In、Out 和 Draw），以及使用它们的客户端程序示例。下面，我们将详细地讨论其基本属性。在很大程度上，Java 的初学者不必知道这些细节。然而，经验丰富的程序员知道，正确理解这些属性有助于编写正确、有效以及高性能的面向对象程序。

下面的表格表述了物体及其名称之间的区别，都是我们熟悉的内容：

类型	典型的对象	典型的名称
网站	本书网站	http://introc.s.princeton.edu
人	计算机科学之父	艾伦·图灵
行星	太阳系的第三颗行星	地球
建筑物	我们的咖啡厅	奥尔登街35号
船	1912年沉没的超级航轮	RMS泰坦尼克号
数字	圆的周长/直径	π
Picture	new Picture("mailrill.jpg")	picture

给定对象可能有多个名称，但每个对象都有自己的标识。我们可以在不改变对象的值的情况下（通过赋值语句）为对象创建一个新的名称，但是当我们更改对象的值（通过调用实例方法）时，所有对象的名称都会引用这个被更改的对象。

下面的比喻可能会帮助读者将这个重要的区别铭记于心。假设我们想将房子重新装潢，于是用铅笔在纸上写下房子的街道地址，然后将它发给一些房屋粉刷匠。现在，如果聘请其中一个粉刷匠粉刷，房子会刷成另一种颜色。任何一张纸上的内容都没有改变，但它们指示的房子已经改变了。其中一位粉刷匠可能会抹去纸上的内容并写下另一间房子的地址，但是改变一张纸上的内容并不会改变其他纸上的内容。Java 引用就像这些纸片：它们持有对象的名称。更改引用不会更改对象，但更改对象会使引用该对象的所有用户都改变。

[362]

著名的比利时艺术家勒内·马格利特（René Magritte）在一幅画中表达出了引用的概念，他创作了一个烟斗的图像，下面是解释文字：ceci n'est pas une pipe（这不是一个烟斗）。我们可以理解说明文字的意思为：画中的烟斗不是真实的烟斗，而只是一个烟斗的图像。也许马格利特的意思是说：说明文字既不是一个烟斗，也不是一个烟斗的图像，仅仅是一个说明文字！在当前的上下文中，这个图强调了一种思想，即一个指向对象的引用仅仅是一个引用，引用不是对象本身。



别名。带有引用类型的赋值语句可以创建引用类型的第二个副本。赋值语句不会创建新的对象，只是对现有对象的另一个引用。这种情况被称为别名：两个变量都指向同一个对象。别名的效果有些出乎意料，因为其与保存内置类型值的变量不同，你一定要正确地理解

它们的不同之处。如果 x 和 y 是内置类型的变量，则赋值语句 $x=y$ 是将 y 的值复制到 x 。而对于引用类型，复制的是引用（而不是值）。

别名是 Java 程序中常见的错误来源，如下例所示：

```
Picture a = new Picture("mandrill.jpg");
Picture b = a;
a.set(col, row, color1); // 一次更新
b.set(col, row, color2); // 再次更新
```

在第二个赋值语句之后，变量 a 和 b 都引用相同的图片对象。改变一个对象的状态会影响引用该对象的别名变量的所有代码。我们习惯于把两个不同的基本类型变量看作两个独立的变量，但是这个直觉不能延续到引用对象。例如，如果前面的代码假定 a 和 b 引用不同的图片对象，则会产生错误的结果。这样的别名错误在对引用对象编程经验不足的人编写的代码中十分常见（这个人常常就是你，所以请注意！）。

不可变类型。正是出于上述原因，我们需要能够定义一些值不能改变的数据类型。如果数据类型值一旦创建就不能改变，那么该数据类型对象是不可变的。不可变的数据类型中所有类型对象都是不可变的。例如，`String` 是一个不可变的数据类型，因为客户程序没有可更改字符串字符的操作接口。与此相对的，可变数据类型指该种类型对象的值被设计为可变的。例如，`Picture` 是可变数据类型，因此我们可以更改其中任意像素颜色。我们会在 3.3 节更详细地讨论不变性。

比较对象。`==` 运算符应用于引用类型时，会检查两个对象引用是否相等（即是否指向同一个对象）。这不同于检查对象是否具有相同的值。例如，请思考以下代码：

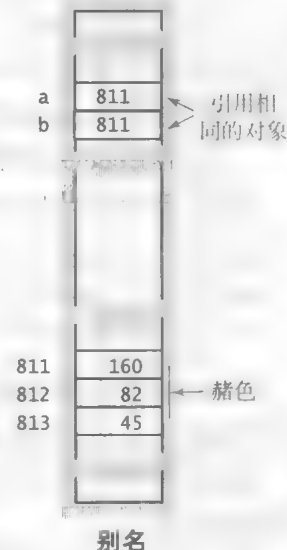
```
Color a = new Color(160, 82, 45);
Color b = new Color(160, 82, 45);
Color c = b;
```

现在 $(a==b)$ 是假的， $(b==c)$ 是真的，但是当你测试颜色是否相同时，我们会想要测试它们的值是否相同——你可能希望测试这三个都相等。Java 没有一个自动的机制来测试对象的值是否相等，这就使得程序员有机会（和责任）定义一个名为 `equals()` 的自定义方法来实现这个功能，如 3.3 节所述。例如，`Color` 类中就有这样一个方法，在这个例子中，`a.equals(c)` 是真的。`String` 类型已经实现了一个 `equals()` 方法，因为经常需要测试两个字符串对象是否具有相同的值（相同的字符序列）。

传值。当调用带有参数的方法时，Java 中的效果就好像每个参数都显示在赋值语句的右侧，而左边是相应的参数名称。也就是说，Java 将来自调用方的参数值的副本传递给方法。如果参数值是基本类型，则 Java 会传递该值的副本；如果参数值是一个对象引用，Java 将传递该对象引用的副本。这种设计被称为传值。

这种设计的一个重要结果是：方法不能直接改变调用方变量的值。对于基本类型，这个策略是我们所希望得到的（两个变量是独立的），但是每次使用引用类型作为方法参数时，我们只是创建了一个别名，因此需要谨慎。例如，如果我们将一个 `Picture` 类型的对象引用

```
Color a;
a = new Color(160, 82, 45);
Color b = a;
```



363

364

传递给方法，该方法不能改变调用方的对象引用（比如使其引用其他图片），但它可以更改对象的值，如调用 `set()` 方法来改变像素的颜色。

数组是对象。在 Java 中，每个非基本数据类型的值都是一个对象。特别地，数组也是对象。与字符串一样，Java 为数组上的一些操作提供特殊的语言支持：声明、初始化和索引。与任何其他对象一样，当将一个数组传递给一个方法时，或在赋值语句的右边使用一个数组变量时，都将复制数组引用，而不是创建数组的副本。数组是可变对象——这个规则适用于我们希望方法能够通过重新排列元素的值来对数组做出更改的情况，例如在 2.1 节中考虑的 `exchange()` 和 `shuffle()` 方法。

对象数组。数组元素可以是任何类型，从 `main()` 函数实现中的 `args[]`（一个字符串数组），到程序 3.1.9 中的 `Out` 对象数组，我们已经使用了多次。当创建一个对象数组时，一般通过两个步骤来完成：

- 通过使用 `new` 和方括号语法创建数组。
- 通过使用 `new` 调用构造函数来创建数组中的每个对象。

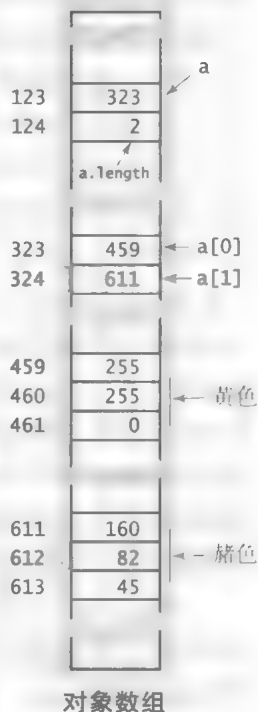
例如，我们将使用下面的代码来创建有两个颜色对象的数组：

```
Color[] a = new Color[2];
a[0] = new Color(255, 255, 0);
a[1] = new Color(160, 82, 45);
```

当然，Java 中的对象数组是对象引用的数组，而不是对象本身。如果对象很大，那么可以通过无须移动而仅仅引用它们来大大提升效率。如果对象很小，那么每次需要获取某些信息时都必须使用引用，效率反而很低。

安全指针。为了提供操作所引用的数据的内存地址的功能，许多编程语言都支持指针（类似于 Java 引用）作为基本数据类型。用指针编程是经常容易出错的部分，这一点已达成共识，所以为指针提供的操作需要精心设计，以帮助程序员避免错误。Java 将这个观点推向了极致（这是许多现代编程语言设计者所青睐的）。在 Java 中，只有一种方法可以创建引用（使用 `new`），并且只有一种方法来操作引用（使用赋值语句）。也就是说，程序员唯一能对引用数据类型做的操作就是创建并复制引用。在编程术语中，Java 引用被称为安全指针，因为 Java 可以保证每个引用都指向特定类型的对象（而不是一个任意的内存地址）。过去编写直接操作指针的代码的程序员认为 Java 根本没有指针，但是人们仍然在讨论非安全指针是否存在。简而言之，当使用 Java 编程时，用户不会直接操纵内存地址，但是如果以后你需要使用其他语言来实现指针的话，请小心！

孤立对象。将不同的对象分配给一个引用变量的功能，可能导致一个程序创建的对象不再被引用的情况。如下图的三个赋值语句。在第三个赋值语句之后，不仅 `a` 和 `b` 引用同一个 `Color` 对象（其 RGB 值为 160、82 和 45），而且最初创建并初始化给 `b` 的对象将不再被任何变量引用。唯一引用该对象的变量是 `b`，但第三条赋值语句覆盖了该引用，结果没有任何引用指向该对象。这样的对象是孤立对象。当变量超出了其作用范围，其指向的对象也会成为孤立对象。Java 程序员一般不会关注孤立对象，因为系统对孤立对象占用的内存会自动复用，接下来将讨论其实现机制。



内存管理。程序常常创建大量的对象，但是在特定时间一般只需要使用其中少量的一部分。因此，程序设计语言和系统需要一种机制：在需要的时候为数据类型值分配内存，并在不需要时（对于某个对象，即在其成为孤立对象的时候）释放内存。对于基本类型，内存管理容易实现，因为编译时就已知所有内存分配所需的信息。Java（以及大多数其他系统）在声明变量时为其在内存空间分配内存，并在变量超出作用范围时释放内存。对象的内存管理更为复杂：Java 知道在创建对象（使用 new）时为对象分配内存，但无法准确知道何时释放对象占用的内存，因为对象是在程序的执行过程中动态地变成孤立对象的，只有那时才能销毁。

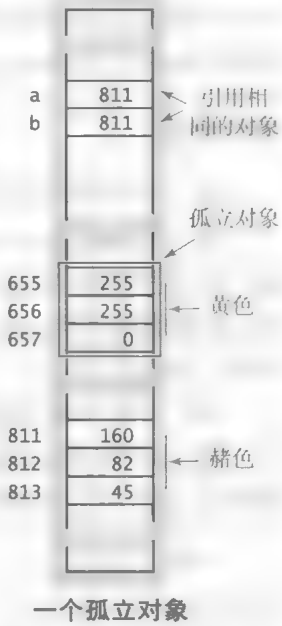
内存泄漏。在许多语言（如 C 和 C++）中，程序员负责分配和释放内存，这样做非常麻烦并且容易出错。例如，假设一个程序释放了一个对象所占据的内存，但是在稍后的程序中继续引用这个对象，而此时，系统可能重新分配了相同的内存以用于其他用途，从而导致各种错误的结果。当程序员无法确保孤立对象的内存已经被释放时，会出现另一个隐患。这种错误被称为内存泄漏，其结果可能导致孤立对象占用的内存空间不断增加（这些内存却不能被重复使用）。这将导致系统性能下降，就像计算机的内存“流走”了一样。读者也许经历过系统响应越来越不灵敏而重新启动计算机的情况，这种行为发生的一个常见原因就是某个应用程序导致了内存泄漏。

垃圾回收。Java 最显著的特点之一是能够自动管理内存。Java 可以跟踪孤立对象，回收其占用的内存空间到空闲的内存池，能让程序员从管理内存的责任中解放出来。这种回收内存的方式被称为垃圾回收。Java 的安全指针策略使其能够高效地自动执行这一操作。程序员仍然在争论“自动垃圾回收的开销”与“不必担心内存管理的便利性”之间的利弊权衡问题。本书给出的结论是：当在 Java 中编程时，你不必编写代码来分配和释放内存，但是如果以后你在其他语言中这样做，请小心！

作为参考，我们在下面的表格中总结了我们在本节中讨论过的例子。选择这些例子是为了帮助读者理解数据类型和面向对象程序设计的基本特征。

一个数据类型是一系列值和定义在这些值上的一组操作的集合。使用基本数据类型，可以处理少量简单的值。字符串（String）、颜色（Color）、图片（Picture）和输入/输出流（In, Out）是高级数据类型，展示了数据抽象的广泛应用。使用一个数据类型时无须理解其具体实现。每种数据类型（Java 库中有数百种，在后续章节中将学习创建自定义数据类型）通过 API（应用程序编程接口）提供其使用信息。客户程序创建对象并赋予对象相应数据类型的值，以及调用实例方法操作这些值。我们使用第 1 章和第 2 章中学到的基本语句和控制结构编写客户程序，但是现在我们有能力处理各种各样的数据类型，而不仅仅是已经习惯的基本数据类型。有了更多的经验，读者会发现使用各种数据类型的能力将拓展程序设计的新视野。

```
Color a, b;  
a = new Color(160, 82, 45);  
b = new Color(255, 255, 0);  
b = a;
```



367

API	描述
Color	颜色
Picture	数字图像
String	字符串
In	输入流
Out	输出流
Draw	绘图

本节中数据类型的总结

比起不使用抽象数据类型的情况，精心设计和实现的数据类型可以使客户程序更清晰简洁，易于开发和维护。本节实现的客户程序有力地证明了这一点。另外，我们将在下一节学习到，实现数据类型其实就是读者已经学习的基本编程技巧的简单应用。特别是解决大型和复杂的应用程序就变成成为应用程序设计适合的数据并定义数据上的操作的过程，然后编写程序的过程就是这些理念的直接体现。一旦学会了这种实现方法，读者可能会惊讶为什么有的程序员在开发大型程序的过程中竟然不使用数据抽象。

[368]

问答环节

问：为什么区分内置（基本）类型和引用类型？

答：因为性能。Java 提供了与内置类型相对应的引用类型 `Integer`、`Double` 等，并且可能会被那些喜欢忽略区别（详情参见 3.3 节）的程序员所使用。内置类型更接近于计算机硬件支持的数据类型，因此使用它们的程序通常运行得更快，并且比使用相应引用类型的程序消耗更少的内存。

问：如果在创建对象时忘记使用 `new`，会发生什么情况？

答：对于 Java 来说，看起来像是你想调用一个带有返回对象类型的静态方法。既然程序没有定义这样一个方法，那么错误信息就和引用一个未定义的符号一样。如果编译如下代码

```
Color sienna = Color(160, 82, 45);
```

将得到这样的错误信息：

```
cannot find symbol
symbol : method Color(int,int,int)
```

构造函数不提供返回值（它们的定义中也没有返回类型），它们只能通过 `new` 关键字调用。如果向构造函数或方法提供了错误的参数数量，则会得到与刚才一样的错误消息。

问：为什么我们可以用函数调用 `StdOut.println(x)` 来打印对象 `x`，而不是 `StdOut.println(x.toString())`？

答：这是一个很好的问题。后者能够工作正常，但是 Java 的机制可以自动调用 `toString()` 方法，为我们省下一些工作。在 3.3 节中，我们将讨论 Java 实现这一机制使用的方法。

问：`=`、`==`、`equals()` 之间的区别是什么？

答：单个等号（`=`）是赋值语句，你一定熟悉这一点。双等号（`==`）是一个二元运算符，用于检查两个操作数是否相同。如果操作数是内置类型，那么当它们具有相同的值时结果为真，否则为假。如果操作数是对象引用，那么当引用同一个对象时，则结果为真，否则为假。也就是说，我们使用 `==` 来测试对象的标识是否相等。Java 的每个数据中都包含一个方法 `equals()`，以便实现为用户提供测试两个对象是否具有相同值的能力。请注意，`(a==b)` 意味着 `a.equals(b)`，反之则不成立。

[369]

问：如何将一个数组作为参数传递给一个函数，并且使得函数无法改变数组中元素的值？

答：没有直接的方法能够实现这个功能——因为数组是可变的。在 3.3 节中，我们将学习如何通过构建封装数据类型并传递该类型的对象引用来实现想要的效果（请参见程序 3.3.3 中的 `Vector`）。

问：如果在创建对象数组时忘记使用 `new`，会发生什么？

答：程序需要使用 `new` 创建每个对象，因此创建一个包含 n 个对象的数组时，需要使用 $n+1$ 次 `new`：对数组使用一次，对 n 个对象每个使用一次。如果忘记创建数组：

```
Color[] colors;
colors[0] = new Color(255, 0, 0);
```

将会得到与尝试将值分配给未初始化的变量时相同的错误消息：

```
variable colors might not have been initialized
  colors[0] = new Color(255, 0, 0);
  ^
```

相反，如果在数组中创建对象时忘记使用 `new`，然后尝试使用它来调用方法

```
Color[] colors = new Color[2];
int red = colors[0].getRed();
```

将会得到一个空指针异常。一般来说，回答这些问题的最好方法是自己编写和编译这些代码，然后尝试解释 Java 的错误消息。这样做可能会帮助我们更快地识别错误。

370

问：请问哪里可以找到有关 Java 如何实现引用和垃圾回收的更多细节？

答：一个 Java 系统可能与另一个完全不同。例如，一个常用的方案是使用一个指针（机器地址）；另一个方案是使用句柄（指向指针的指针）。前者提供更快速的数据访问；后者便于垃圾回收。

问：为什么三原色是红、绿、蓝而不是红、黄、蓝？

答：理论上讲，任何三种包含每个主颜色某些分量的颜色都可以起作用，但目前为止，广泛使用的两种不同的颜色模型：一种（RGB）在电视屏幕、计算机显示器和数码相机上可产生良好的色彩，另一种（CMYK）通常用于印刷（参见练习 1.2.32）CMYK 包括黄色（青色、品红色、黄色和黑色）。两种不同的颜色模型都有各自的实用性，对于印刷，因为印刷油墨吸收颜色；当存在两种不同颜色的油墨时，吸收的颜色越多反射的颜色越少。与之相对的，视频显示器会发出颜色，所以存在两个不同颜色的像素时，会反射更多的颜色。

问：`import` 语句的目的究竟是什么？

答：很简单：它只是为了让你少输入一些字。例如，在程序 3.1.2 中，它使用户可以在代码中的任何地方使用 `Color`，而不用输入 `java.awt.Color`。

问：在 Grayscale 程序（程序 3.1.4）中，分配和释放数千个 `Color` 对象是否有任何问题？

答：所有的编程语言结构都有一定的代价。这种情况下的开销是合理的，因为分配 `Color` 对象的时间与绘制图像的时间相比是微小的。

371

问：为什么 `String` 方法调用 `s.substring(i, j)` 返回从索引 i 开始并以 $j-1$ （而不是 j ）结束的 `s` 的子串？

答：为什么数组 `a[]` 的索引从 0 到 `a.length()-1`，而不是从 1 到长度？编程语言设计师做出选择；我们接受这个规则。这个规则的一个很好的结果是，提取的子串的长度是 $j-i$ 。

问：通过值传递参数和通过引用传递参数之间有什么区别？

答：通过值传递，当你调用一个带参数的方法时，每个参数都被计算出来，并将结果值副本传递给该方法。这意味着，如果方法直接修改参数变量，那么这个修改对于调用者是不可见的。通过引用传递，每个参数的内存地址被传递给方法。这意味着如果一个方法修改了一个参数变量，这个修改对于调用者是可见的。从技术上讲，Java 是一种纯粹的按值传参语言，其中的值可以是内置类型值或对象引用。因此，当你将一个内置类型值传递给方法时，

该方法将无法修改调用方法中的相应值；当你传递一个方法的对象引用时，方法不能修改对象引用（比如说引用一个不同的对象），但是其可以改变底层对象（通过使用对象引用来调用对象的一个方法）。出于这个原因，一些 Java 程序员使用术语“通过对象引用传递”来表示 Java 的引用类型参数传递的方法。

问：String 和 Color 中 equals() 方法的参数是 Object 类型的。参数不应该分别是 String 类型和 Color 类型吗？

答：不。在 Java 中，equals() 是一个特殊的方法，它的参数类型应该始终是 Object。这是 Java 用于支持 equals() 方法的继承机制的设计，我们将在 3.3 节讨论这一点。现在，我们可以放心地忽略这种区别。

问：为什么图像处理数据类型名为 Picture 而不是 Image？

372

答：因为已经有一个名为 Image 的内置 Java 库。

练习

- 3.1.1 请编写一个程序，程序需要输入一个 double 类型的命令行参数 w，然后创建 4 个带电荷值为 1.0 的 Charge 对象，分别位于位置点 (0.5,0.5) 的上、下、左、右四个方向上，与 (0.5, 0.5) 的距离为 w，请输出位置点 (0.25,0.5) 的电势。
- 3.1.2 编写一个程序，程序命令行获取 0 到 255 之间的 3 个整数，分别代表一种颜色的红色、绿色和蓝色值，然后创建并显示一幅 256×256 图像，用该颜色填充图像中的像素。
- 3.1.3 修改程序 AlbersSquares (程序 3.1.2)，实现如下功能：程序须输入 9 个命令行参数，指定 3 种颜色，然后绘制 6 个正方形，标识出所有的 Albers 正方形，其中，大的正方形采用一种颜色绘制，小的正方形则采用另一种颜色绘制。
- 3.1.4 请编写一个程序，程序从命令行参数读入一个文件名，用于指定一个灰度图像文件，使用 StdDraw 绘制该灰度图像 256 个灰度值出现频率的直方图。
- 3.1.5 编写一个程序，以图像文件的名称为命令行参数，实现图像水平翻转。
- 3.1.6 编写一个程序，将图像文件的名称作为命令行参数，创建 3 个图片对象，一个只包含原始图像的红色分量，一个只包含绿色分量，另一个只包含蓝色分量，并显示三幅图片。
- 3.1.7 编写一个程序，将图像文件的名称作为命令行参数，计算出该图像中能够框住所有非白色像素的最小包围框（假设包围框是边平行于 x 轴和 y 轴的矩形），输出该框的左下角和右上角的像素坐标。
- 3.1.8 编写一个程序，程序需要输入两个命令行参数：一个图像文件名和位于图像之内的一个矩形的像素坐标。从标准输入读取若干 Color 值（颜色值使用 3 个整数表示）。请设计一个过滤器，从输入的 Color 值中，筛选并输出与矩形框中所有的前景色 / 背景色相兼容的颜色（此类过滤器可用于为文本选择一种颜色以标记一幅图像）。
- 3.1.9 请编写一个函数 isValidDNA()，输入一个字符串参数，当且仅当字符串参数完全由字符 A、C、T 和 G 组成时，返回 True。
- 3.1.10 请编写一个函数 complementWC()，输入一个 DNA 字符串参数，返回其 Watson-Crick 碱基互补，即 A 和 T 互换、C 和 G 互换。
- 3.1.11 请编写一个函数 isWatsonCrickPalindrome()，输入一个 DNA 字符串参数，如果 DNA 字符串是 Watson-Crick 互补碱基回文的形式，程序返回 True，否则返回 False。Watson-Crick 互补碱基回文是一个 DNA 序列，它与 Watson-Crick 互补碱基序列的逆序形式相同。

373

3.1.12 请编写一个程序，检查一个 ISBN 号是否有效（详情参见练习 1.3.35）。请注意 ISBN 号可能会在任意位置出现连字符（-）。

3.1.13 下面代码片段的输出是什么？

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
StdOut.println(string1);
StdOut.println(string2);
```

3.1.14 下面代码片段的输出是什么？

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

答案：“Hello World”。字符串对象是不可变对象，每个字符串方法都会返回一个新的 String 对象以及合适的值（但不会改变用于调用方法的原字符串的值）。此代码忽略返回的对象，只是打印原始字符串。要打印“WORLD”，用 `s=s.toUpperCase()` 和 `s= s.substring(6,11)` 替换第二个和第三个语句。

374

3.1.15 对于一个字符串 s 和一个字符串 t ， t 中的字符首尾相接，经过若干次循环移动后能够与 s 匹配，那么就说 s 是 t 的循环移位。例如，ACTGACG 是 TGACGAC 互为循环移位。检测这种情况在基因组序列的研究中是重要的。编写一个程序，检查两个给定的字符串 s 和 t 是否互为循环移位。提示：解决方案中可能要同时使用 `indexOf()` 和字符串连接。

3.1.16 给定代表一个网站 URL 的字符串，请编写代码片段以确定其顶级域名。例如，本书官网 `cs.princeton.edu` 的顶级域名是 `edu`。

3.1.17 编写一个以域名作为参数的静态方法，返回其反向域名（颠倒点之间的字符串顺序）。例如，`cs.princeton.edu` 的反向域名是 `edu.princeton.cs`。此计算对于 Web 日志分析非常有用（见练习 4.2.36）。

3.1.18 下面的递归函数返回什么？

```
public static String mystery(String s)
{
    int n = s.length();
    if (n <= 1) return s;
    String a = s.substring(0, n/2);
    String b = s.substring(n/2, n);
    return mystery(b) + mystery(a);
}
```

3.1.19 为 PotentialGene（程序 3.1.1）写一个测试用户程序，它使用字符串作为命令行参数，并检测它是否是一个潜在基因。

3.1.20 写另一个版本的 PotentialGene（程序 3.1.1），在一个长的 DNA 字符串中找到包含在子字符串中的所有潜在基因。添加一个命令行参数以允许用户指定潜在基因的最小长度。

3.1.21 编写一个从输入流中读取文本并将其输出到输出流的过滤器，删除仅包含空白的行。

375

3.1.22 编写一个程序，该程序将一个起始字符串和一个结束字符串作为命令行参数，并打印以起始字符串开始、结束字符串结束的所有子字符串。注意：考虑重叠的情况！

3.1.23 修改 StockQuote（程序 3.1.8），以支持在命令行上输入多个股票号码。

- 3.1.24 用于 Split (程序 3.1.9) 的示例文件 DJIA.csv 列出了其自记录以来每天道琼斯股票市场平均价格的日期、最高价、交易量和最低价。从本书官网下载这个文件, 编写一个程序, 创建两个 Draw 对象, 一个用于价格, 另一个用于数量, 并以从命令行接收的参数比例绘制股票价格和成交量图。
- 3.1.25 编写一个程序 Merge, 程序从命令行接收若干参数: 一个分隔符字符串和任意数量的文件名。以指定的字符串作为分隔符, 拼接每个文件相应的行的内容, 并将结果写入标准输出, 从而实现 Split (程序 3.1.9) 的相反操作。
- 3.1.26 查找一个发布本地区当前气温的网站, 并写一个 Web 信息抓取程序 Weather, 输入 “java weather” 后跟邮政编码, 可获取邮政编码所在地区的天气预报。
- 3.1.27 假设 `a[]` 和 `b[]` 都是由数百万个整数组成的整数数组。下面的代码是做什么的, 需要多长时间?

```
int[] temp = a; a = b; b = temp;
```

答案: 它交换数组, 但它通过复制对象引用来实现这一功能, 所以不需要拷贝数百万个值。

- 3.1.28 描述以下函数的效果。

```
public void swap(Color a, Color b)
{
    Color temp = a;
    a = b;
    b = temp;
}
```

376

创新练习

- 3.1.29 图片文件格式。编写一个静态方法库 RawPicture, 提供 `read()` 和 `write()` 方法用于读取和保存文件中的图片。`write()` 方法将一个图片对象和一个文件的名称作为参数, 将图片转换为指定的文件写入, 使用以下格式输出: 如果图片是 $w \times h$, 则输出 w , 然后 h , 最后输出 $w \times h$ 个代表像素颜色值的整数三元组, 按行优先顺序输出。`read()` 方法将图片文件的名称作为参数, 并通过从指定文件中读取数据来返回一个图片对象, 格式如上所述。注意: 所占用的磁盘空间比图像文件要大得多, 因为标准图像文件格式通常会压缩这些信息, 所以不会占用太多的空间。
- 3.1.30 声音可视化。使用 StdAudio 和 Picture 对象编写一个程序, 在播放音乐的同时创建一个有趣的二维颜色可视化文件。请读者发挥自己的创意!
- 3.1.31 Kamasutra 密码。编写一个过滤器程序 KamasutraCipher, 它将两个字符串作为命令行参数 (密钥字符串), 然后从标准输入中读取若干字符串 (用空格分隔), 按照密钥字符串指定替换规则替换输入的每个字母, 并将结果输出到标准输出。这个操作是已知最早的密码系统的基础。密钥字符串的条件是它们必须具有相同的长度, 并且标准输入中的所有字母必须包含在其中一个字符串中。例如, 如果两个密钥字符串是 THEQUICKBROWN 和 FXJMPSVRLZYDG, 那么我们得到表格

```
T H E Q U I C K B R O W N
F X J M P S V L A Z Y D G
```

这张表告诉我们, 当把标准输入过滤到标准输出时, 应该用 F 代替 T、T 代替 F、H 代替 X、X 代替 H 等。通过将每个字符替换为对应字符, 实现信息的加密。例如, 消息 “MEET AT ELEVEN” 被编码为 “QJFF BF JKJCJG”。接收消息的人可以使用相同的密钥来解密明文。

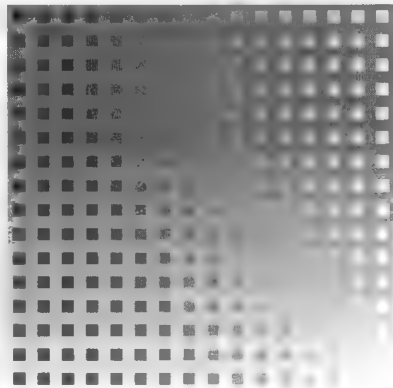
377

3.1.32 验证密码的安全性。编写一个将字符串作为参数的函数，如果满足以下条件则返回真，否则返回假：

- 至少 8 个字符。
- 包含至少一个数字 (0~9)。
- 包含至少一个大写字母。
- 包含至少一个小写字母。
- 包含至少一个既不是字符也不是数字的字符。

类似的检测通常用于 Web 密码验证。

3.1.33 颜色研究。编写一个程序，实现右图颜色研究结果的显示。图中展示了 Albers 正方形，对应 256 级别蓝色（每行从左到右按照从蓝色到白色渐变）和灰度（每列从上到下按从黑色到白色渐变），而这些也正是印刷本书英文原书时使用的颜色。



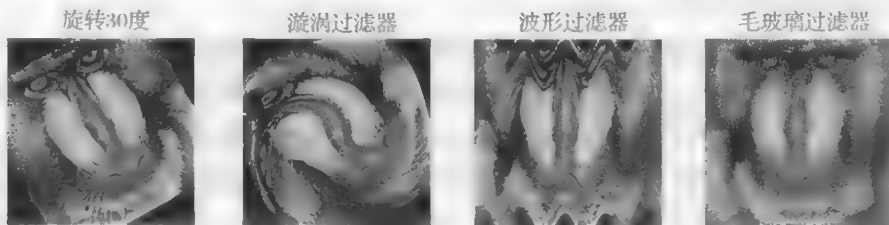
一个颜色研究

3.1.34 熵。香农熵用于表示输入数据的信息量，在信息理论和数据压缩中起着基石的作用。给定一个包含 n 个字符的字符串，设 f_c 为字符 c 出现的频率。 $p_c = f_c/n$ 是 c 在一个随机的字符串中的频率的估计，信息熵定义为字符串中所有字符的 $p_c \log_2 p_c$ 之和。信息熵被用于度量一个字符串的信息量：如果每个字符出现次数相同，则信息熵为最小值。编写一个程序，将一个文件名作为命令行参数，并输出该文件的熵。

分别针对你经常阅读的一个网页、最近撰写的一篇文章、从网站上查找到的果蝇基因组，运行并测试该程序。

378

3.1.35 平铺。编写一个程序，该程序以图像文件的文件名和两个整数 m 和 n 作为命令行参数，并创建一个 $m \times n$ 的图像平铺。



图像过滤器

3.1.36 旋转过滤器。编写一个程序，其中采用两个命令行参数（图像文件名称和一个实数 θ ），并将图像逆时针旋转 θ 角度。为了实现旋转，将源图像中每个像素 (s_i, s_j) 的颜色复制到目标像素，目标像素的坐标 (t_i, t_j) 由以下公式给出：

$$t_i = (s_i - c_i) \cos \theta - (s_j - c_j) \sin \theta + c_i$$

$$t_j = (s_i - c_i) \sin \theta - (s_j - c_j) \cos \theta + c_j$$

其中 (c_i, c_j) 是图像的中心点。

3.1.37 漩涡过滤器。创建一个漩涡效果。类似于旋转，但不同之处在于，角度的改变是像素到中心距离的函数。使用与前面练习中相同的公式，但 θ 是 (s_i, s_j) 的函数： θ 的值是 $(\pi/256)$ 乘以该像素到中心点的距离。

3.1.38 波形过滤器。通过将源图像中每个像素 (s_i, s_j) 的颜色复制到一个目标像素 (t_i, t_j) 中，写出一个像前两个练习中那样的过滤器，从而产生波浪效果，其中 $t_i = s_i$ 和 $t_j = s_j + 20 \sin(2\pi s_j / 64)$ 。添加代码以将振幅（图中为 20）和频率（图中为 64）作为命令行参数。用不同的参数值运行测试程序。

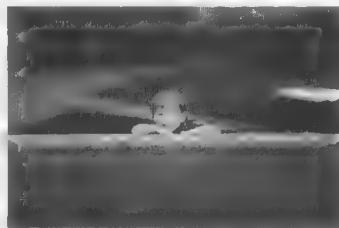
3.1.39 毛玻璃过滤器。编写一个程序，将图像文件的名称作为命令行参数，并应用毛玻璃过滤器：将每个像素 p 设置为随机相邻像素的颜色（邻居像素的坐标点位置距离 p 的坐标点位置在 5 个像素之内）。

3.1.40 幻灯片放映。编写一个程序，将多个图像文件的名称作为命令行参数，并以幻灯片的形式显示（每两秒一次），在图像之间使用黑色淡入淡出效果，即图像退出时褪色为黑色，然后由黑色褪色为下一张图像。

3.1.41 变形。本书中讲到的 Fade 程序的示例图像在垂直方向上没有对齐（mandrill 中嘴的位置比 Darwin 中嘴的位置低很多）。给 Fade 程序在垂直维度中添加一个变化，使得过渡的效果更加平滑。

3.1.42 数码变焦。编写程序 Zoom，将图像文件的名称和三个数字 s 、 x 和 y 作为命令行参数，并显示输入图像某一部分的放大结果。数字范围都在 0 和 1 之间，其中 s 为缩放的比例因子， (x, y) 作为输出图像中心点的相对坐标。使用此程序可放大计算机上亲属或宠物的照片（如果你的照片来自旧手机或相机，照片放得太大后就会出现很多不自然的瑕疵）。

```
% java Zoom boy.jpg 1 .5 .5
```



© 2014 Janine Dietz

```
% java Zoom boy.jpg .5 .5 .5
```



```
% java Zoom boy.jpg .2 .48 .5
```



数码变焦

3.2 创建数据类型

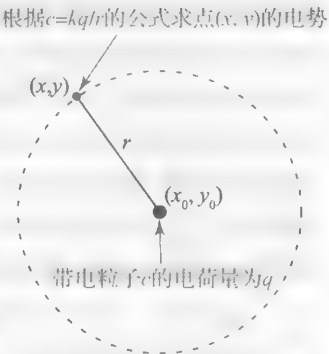
原则上，仅使用内置的 8 种基本数据类型就可以编写所有的程序。但是，正如上一节所述，在更高的抽象层次上编写程序会更加便利。因此，Java 的语言标准和扩展库中定义了各种数据类型。尽管如此，我们肯定不能指望 Java 包含了所有我们可能使用的数据类型，所以能够定义自己的数据类型就显得十分必要。本节介绍如何使用 Java 的类（class）构建数据类型。

使用 Java 类实现数据类型，与实现一个包含若干函数的静态方法库并没有很大区别，主要区别在于我们这次需要将方法实现与数据相关联。API 指定了需要实现的构造函数和实例方法，但是用户可以自由选择任何方便的方式来实现。为了巩固基本概念，我们首先实现 3.1 节中介绍的带电粒子的数据类型。接下来，我们通过一系列示例（从复数到股票账户，包括后续章节中将使用的一些软件工具）来说明创建数据类型的过程。实用的客户代码是任何数据类型的价值的最好证明，所以我们将讨论一些客户程序示例，包括描述著名而迷人的 Mandelbrot 集合。

定义一个数据类型的过程被称为数据抽象。我们关注数据并实现对这些数据的操作。在计算中应当清晰地分离数据和相关的计算任务。对物理对象或者熟悉的数学概念进行抽象建模是简单而且实用的，但是数据抽象的强大之处在于它允许我们对任何可以精确描述的东西进行建模。读者一旦熟悉这种编程风格的经验，就会发现数据抽象的使用可以帮助我们解决很多复杂的编程挑战。

数据类型的基本元素 为了描述在 Java 类中实现数据类型的过程，我们将讨论带电粒子的数据类型 Charge。特别是，我们感兴趣的是采用库仑定律的二维模型，即一个带电粒子在给定位置点的电势可表示为 $V=kq/r$ ，其中 q 是电荷量， r 是从位置点到电荷的距离， $k=8.99 \times 10^9 \text{ N} \cdot \text{m}^2/\text{C}^2$ ，是静电常数。当存在多个带电粒子时，任何一点的电势等于各个带电粒子产生的电势之和。为了保持一致性，我们使用 SI 国际单位制：在公式中， N 表示牛顿（力）， m 表示米（距离）， C 表示库伦（电荷）。

API。应用程序编程接口是与所有客户程序之间的协议，因此是所有实现的起点。以下是带电粒子类型的 API：



平面中带电粒子的库仑定律

```
public class Charge
{
    Charge(double x0, double y0, double q0)
    double potentialAt(double x, double y)  (x, y) 处的电势
    String toString()                        字符串表示
}
```

带电粒子的 API (参见程序 3.2.1)

为了实现 Charge 数据类型，我们需要定义数据类型的值，并实现创建带电粒子的构造函数，方法 potentialAt() 用于返回电荷在给定位置点 (x, y) 处的电势，toString() 方法返回电荷的字符串表示形式。

类。在 Java 中，我们把一个数据类型实现为一个类 (class)。同使用的静态方法库一样，可以把一个数据类型的代码存储在一个单独的文件中，后跟 .java 扩展名。我们已经实现过 Java 类，但是我们实现的类并没有数据类型的关键特性：构造函数、若干实例变量和若干实例方法。构建的这些模块都通过访问（或可见）修饰符进行限定。接下来我们讨论这四个概念，并举例说明，最终实现 Charge 数据类型（程序 3.2.1）。

383

访问修饰符。在类名称、实例变量名称和方法名称之前的关键字 public、private 和 final 被称为访问修饰符。public 和 private 修饰符限制客户代码的访问：我们将类中的每个实例变量和方法指定为 public(客户可访问) 或者 private(客户不可访问)。最后一个修饰符 (final) 表示变量的值一旦被初始化就不能更改——它是只读的。我们对修饰符的使用规则是，API 中的构造函数和方法使用 public 修饰符（因为规则承诺将它们提供给客户），而其他的都使用 private 修饰符。通常，私有方法是用于简化类中其他方法的代码的辅助方法。Java 对修饰符的使用没有太严格的限制，3.3 节中我们再讨论这些规则制定的原因。

实例变量。编写代码实现操作数据类型的实例方法，首先需要声明实例变量，实例变量用于在代码中存储相关变量的值。一个变量属于一个类型的特定实例。使用与声明局部变量相同的方式声明实例变量，包括类型和名称：Charge 包含 3 个 double 型实例变量——两个用来描述带电粒子在平面中的位置，一个用来描述带电粒子的电量。这些声明由类中的第一个语句创建，而不是在 main() 或任何其他方法中。与在方法或模块内定义的局部变量相比，实例变量的定义存在一个重要的区别：在给定时间内，每个局部变量只能对应于一个值，但是存在与实例变量对应的许多值（每个对象都是数据类型的一个实例，每个实例中都有一个实例变量的副本）。这种设计不会产生歧义，因为每次调用一个实例方法时，都是使用对象引用来实现的——被引用的对象的实例变量就是正在操作的值。

构造函数。构造函数用于创建一个对象并返回该对象的引用。Java 客户程序使用关键字

new 自动调用构造函数。Java 可以自动完成大部分工作：用户只需要编写代码将实例变量初始化为有意义的值。构造函数名总是与类名相同，但是可以重载构造函数，并拥有多个构造函数，每个构造函数拥有不同的参数，就像静态方法一样。

对于客户程序而言，关键字 **new** 与构造函数（括号内带有参数）形成的组合，就像一个返回指定类型对象引用的函数调用一样。构造函数没有返回类型，因为构造函数总是返回对其数据类型对象的引用（数据类型、类和构造函数都是相同的名称）。每次客户程序调用构造函数时，Java 会自动实现以下功能：

- 为对象分配内存。
- 调用构造函数代码来初始化实例变量。
- 返回对新创建对象的引用。

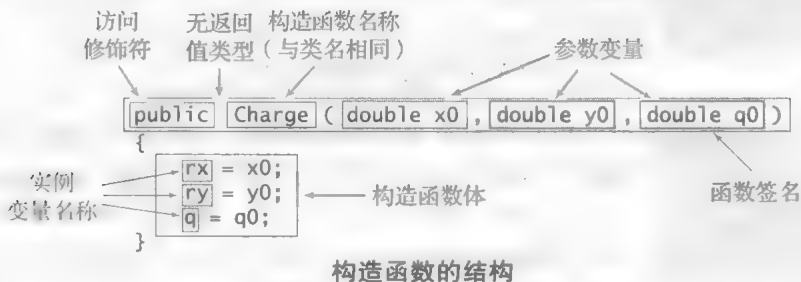
Charge 的构造函数是 Java 中构造函数的经典样式：使用客户程序提供的值初始化实例变量。

```
public class Charge
{
    private final double rx, ry;
    private final double q;
    ...
}
```

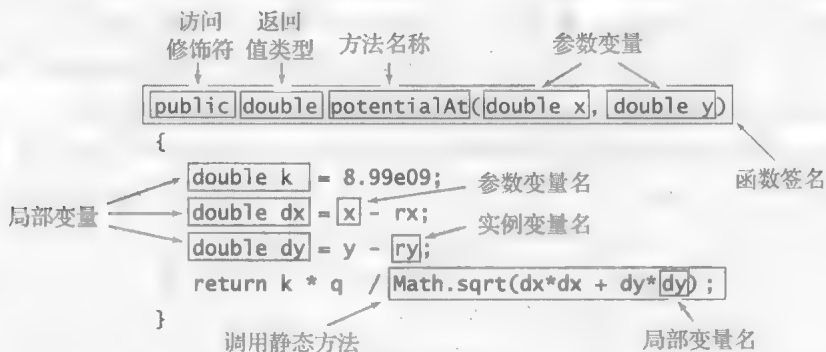
实例变量的声明

访问修饰符

实例变量



实例方法。在实现实例方法时，我们编写的代码与第2章中实现静态方法（函数）的代码十分相似。每个方法都有一个签名（签名指定了方法的返回类型以及参数变量的类型和名称）和一个主体（由一系列语句组成，包括一个返回语句，返回值返回给客户程序）。当客户程序调用一个实例方法时，系统用客户程序提供的值初始化参数变量，接着执行程序中的语句直到 **return**，这时将计算结果返回给客户程序，就像客户程序中的方法调用语句被替换成了返回函数的结果一样。以上这些都与静态方法相同，但实例方法有一个关键的不同之处：它们可以对实例变量进行操作。



方法内的变量。对应的，我们编写的 Java 代码使用以下三种变量实现实例方法：

- 参数变量
- 局部变量

● 实例变量

前两种变量与静态方法相同：参数变量在方法的声明中指定，调用方法时用客户端提供的
数据初始化变量，局部变量在函数体中声明和初始化。参数变量的作用范围是整个方法，局部
变量的作用范围在它们被定义的模块语句中。实例变量则完全不同：它们为类中的对象保存数
据类型的值，其作用域是整个类。如何确定我们想要使用哪个对象的值？只要想一想这个
问题，很容易得到答案。类的每个对象都有一个值：类方法中的代码引用了用于调用方法的对
象的值。例如，当我们写 `c1.potentialAt(x, y)` 时，`potentialAt()` 中的代码引用 `c1` 的实例变量。

`Charge` 中 `potentialAt()` 的实现使用了以上所有三种变量，如“实例方法的结构”图所
示，在下表中进行汇总。

你一定要清楚地理解实现实例方法中三种类型变量之间的区别。理解这些差异是面向对
象程序设计的关键。

变量名称	变量用途	应用示例	作用范围
实例变量	数据类型的值	<code>rx, ry</code>	类中
参数变量	将参数从客户程序传递到方法	<code>x, y</code>	方法中
局部变量	方法中临时使用	<code>dx, dy</code>	模块中

实例方法中的变量

386

程序3.2.1 带电粒子

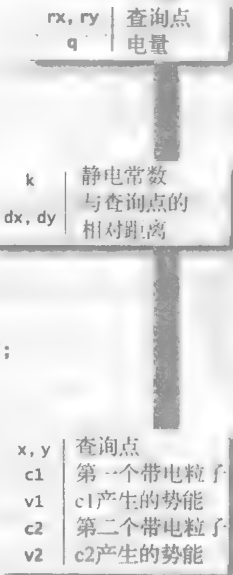
```
public class Charge
{
    private final double rx, ry;
    private final double q;

    public Charge(double x0, double y0, double q0)
    { rx = x0; ry = y0; q = q0; }

    public double potentialAt(double x, double y)
    {
        double k = 8.99e09;
        double dx = x - rx;
        double dy = y - ry;
        return k * q / Math.sqrt(dx*dx + dy*dy);
    }

    public String toString()
    {
        return q + " at " + "(" + rx + ", " + ry + ")";
    }

    public static void main(String[] args)
    {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        Charge c1 = new Charge(0.51, 0.63, 21.3);
        Charge c2 = new Charge(0.13, 0.94, 81.9);
        StdOut.println(c1);
        StdOut.println(c2);
        double v1 = c1.potentialAt(x, y);
        double v2 = c2.potentialAt(x, y);
        StdOut.printf("%.2e\n", (v1 + v2));
    }
}
```



带电粒子数据类型的实现包含每个数据类型的基本组成部分：实例变量 `rx`、`ry` 和 `q`，构造函数 `Charge()`，实例方法 `potentialAt()` 和 `toString()`，以及一个测试客户程序 `main()`。

```
% java Charge 0.2 0.5
21.3 at (0.51, 0.63)
81.9 at (0.13, 0.94)
2.22e+12
```

```
% java Charge 0.51 0.94
21.3 at (0.51, 0.63)
81.9 at (0.13, 0.94)
2.56e+12
```

387

测试客户程序。每个类都可以定义自己的测试程序 `main()`，通常 `main()` 方法会保留以测试数据类型。测试客户程序至少应该调用类中的每个构造函数和实例方法。例如，程序 3.2.1 中的 `main()` 方法需要两个命令行参数 `x` 和 `y`，创建两个 `Charge` 对象，并且输出这两个带电粒子在位置 (x, y) 处产生的总电势。当存在多个带电粒子时，任何一点的电势等于各个带电粒子产生的电势之和。

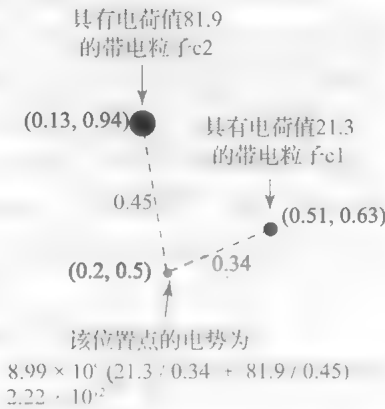
这些是我们需要了解的基本内容，以便能够在 Java 中自定义数据类型。在我们将开发的每种数据类型（Java 类）中，都要具有相同的基本元素：实例变量、构造函数、实例方法和测试客户程序。每种数据类型的开发均遵循相同的步骤。为了完成一个计算目标，我们不应该纠结于下一步需要采取什么具体行动（开始学习编程时经常这样做），而应该考虑客户程序的需求，然后在数据类型中实现这些需求。

创建数据类型的第二步是设计其 API。API 的目的是将客户端与具体的实现分开，以便实现模块化编程。设计 API 时有两个目标。首先，我们希望客户端代码清晰和正确。事实上，在最终确定 API 之前，先编写一些客户代码是一个好主意，以确保设计的数据类型操作符合客户程序的需求。其次，我们必须能够实现这些运算操作。很显然，无法实现的运算操作是毫无意义的。

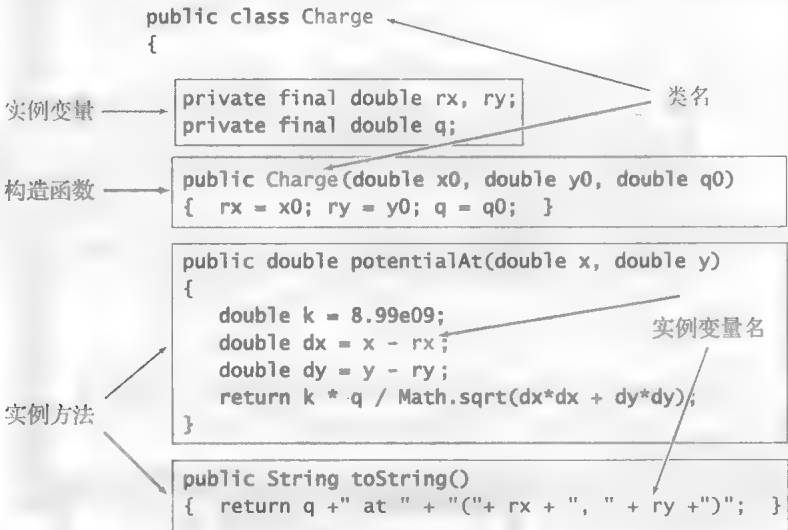
创建一个数据类型的第二步是实现满足其 API 的 Java 类。首先编写构造函数以定义和初始化实例变量，然后编写处理实例变量的方法以实现所需要的功能。

创建数据类型的第三步是编写一个测试客户程序，以验证前两个步骤中做出的设计是否正确。

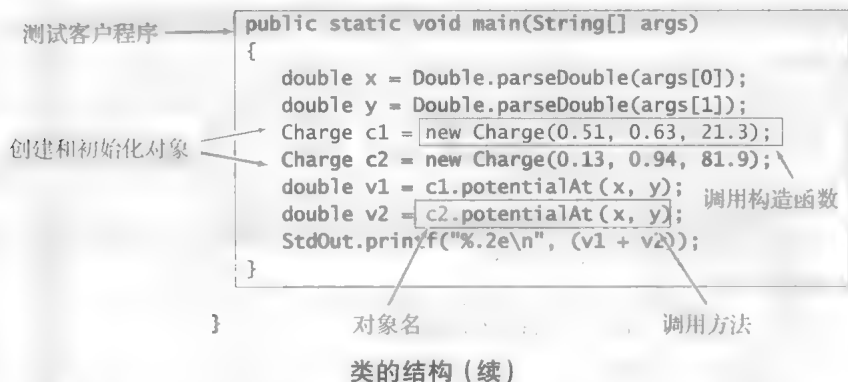
定义数据类型的值是什么？针对这些值，客户程序会执行哪些操作？确定了这些基本问题后，我们就可以创建新的数据类型，然后像使用内置数据类型一样，使用自定义数据类型来编写客户应用程序。本节末尾将提供许多练习，旨在为读者提供创建数据类型的经验。



388



类的结构



389

秒表 面向对象程序设计的特点之一是，通过创建抽象编程对象，就可以轻松地对真实世界的对象建模。作为一个简单的程序示例，Stopwatch（程序 3.3.2）实现了以下 API：

```
public class Stopwatch
```

```
    Stopwatch()
```

创建一个新的秒表对象并运行

```
    double elapsedTime()
```

自秒表创建以来所经过的时间（以秒为单位）

Stopwatch API（见程序 3.2.2）

简单来说，Stopwatch 对象是老式秒表的精简版本。创建一个 Stopwatch 对象后，秒表开始运行，我们可以通过调用 elapsedTime() 方法来查询秒表已经运行的时间。用户也可以加入各种各样、花里胡哨的功能到 Stopwatch——任何你能够想象到的东西。你是否希望重置秒表？开始或停止秒表？再加入一个单圈计时器？添加这些功能很容易的（具体参见练习 3.2.12）。

Stopwatch 的实现使用了 Java 的系统方法 System.currentTimeMillis()，其返回一个 long 型值，以毫秒为单位（自 1970 年 1 月 1 日凌晨 0 点整以来的毫秒数）。Stopwatch 数据类型的实现非常简单。一个 Stopwatch 对象保存其创建时间到一个实例变量中，每当客户端调用其 elapsedTime() 方法时，都会返回该时间与当前时间的时间差值。Stopwatch 对象本身并不计时（计算机内部系统时钟为所有的 Stopwatch 对象计时），Stopwatch 对象只是造成了它为客户程序计时的幻觉。为什么客户程序不直接使用 System.currentTimeMillis()？我们当然可以这样做，但使用 Stopwatch 使得客户程序更容易理解和维护。



老式秒表

测试客户程序是一个典型实现。它首先创建两个 Stopwatch 对象，然后使用它们来测量两种不同计算的运行时间，最后输出运行时间。自最初运行的若干程序开始，就一直存在一个难题：在程序开发中，一种方法是否优于另一种方法？这个问题至关重要。在 4.1 节中，我们将开发一个科学的方法来理解计算的开销。Stopwatch 在该方法中是有用的工具。

390

直方图 现在，我们讨论一个数据类型，这个类型可以使用我们熟悉的直方图显示数据。为简单起见，假定数据是 0 到 n 之间的整数值序列。直方图计算每个值的出现次数，并给每个值绘制条形图（高度与其出现频率成正比）。以下 API 描述了这些操作：

程序3.2.2 秒表

```

public class Stopwatch
{
    private final long start;
    public Stopwatch()
    { start = System.currentTimeMillis(); }
    public double elapsedTime()
    {
        long now = System.currentTimeMillis();
        return (now - start) / 1000.0;
    }
    public static void main(String[] args)
    {
        // 使用Math.sqrt()计算时间
        int n = Integer.parseInt(args[0]);
        Stopwatch timer1 = new Stopwatch();
        double sum1 = 0.0;
        for (int i = 1; i <= n; i++)
            sum1 += Math.sqrt(i);
        double time1 = timer1.elapsedTime();
        StdOut.printf("%e (%.2f seconds)\n", sum1, time1);

        // 使用Math.pow()计算时间
        Stopwatch timer2 = new Stopwatch();
        double sum2 = 0.0;
        for (int i = 1; i <= n; i++)
            sum2 += Math.pow(i, 0.5);
        double time2 = timer2.elapsedTime();
        StdOut.printf("%e (%.2f seconds)\n", sum2, time2);
    }
}

```

这个类实现了一个简单的数据类型，可以用于比较性能关键（performance critical）方法的运行时间（具体参见4.1节）。测试客户程序比较Java数学库中两个函数计算平方根的运行时间。对于计算从1到*n*的数字的平方根之和的任务，调用Math.sqrt()比调用Math.pow()快10倍以上。结果可能因系统而异。

```

% java Stopwatch 100000000
6.666667e+11 (0.65 seconds)
6.666667e+11 (8.47 seconds)

```

```
public class Histogram
```

```
    Histogram(int n)    创建0到n-1整数值的直方图
```

```
    double addDataPoint(int i)  添加整数i的出现次数
```

```
    void draw()          在标准绘图上绘制直方图
```

直方图的API（见程序3.2.3）

要实现这样一个数据类型，我们必须首先确定使用什么样的实例变量。在这种情况下，需要使用一个数组作为实例变量。具体来说，Histogram（程序3.2.3）维护一个实例变量freq[]，对于0到n-1之间的每个i，便于freq[i]记录数据值在数据中出现的次数。Histogram类型还包含一个整数实例变量max，用于存储所有值中的最大频率（对应于最高条柱的高度）。实例方法draw()使用变量max来设置标准绘图窗口的纵轴尺度，并调用方法StdStats.plotBars()绘制数据值的直方图。main()方法是执行伯努利试验的示例客户程序。它比Bernoulli（程序2.2.6）简单得多，因为其使用Histogram数据类型。

通过创建像 Histogram 这样的数据类型，我们可以体会到第 2 章讨论的模块化编程（可复用代码、独立开发小程序等）的好处，以及数据分离的优点。如果没有 Histogram 类型，我们必须将用于创建直方图的代码与用于计算数据的代码混合起来，结果会导致程序比两个单独的程序更加难以理解和维护。在计算中应当清晰地分离数据和相关的计算任务。

391
392

程序3.2.3 直方图

```
public class Histogram
{
    private final double[] freq;
    private double max;

    public Histogram(int n)
    { // 创建一个新的直方图
      freq = new double[n];
    }

    public void addDataPoint(int i)
    { // 整数i的出现次数增加1
      freq[i]++;
      if (freq[i] > max) max = freq[i];
    }

    public void draw()
    { // 绘制（和缩放）直方图
      StdDraw.setYscale(0, max);
      StdStats.plotBars(freq);
    }

    public static void main(String[] args)
    { // 见程序2.2.6
      int n = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);
      Histogram histogram = new Histogram(n+1);
      StdDraw.setCanvasSize(500, 200);
      for (int t = 0; t < trials; t++)
        histogram.addDataPoint(Bernoulli.binomial(n));
      histogram.draw();
    }
}
```



此数据类型支持使用简单的客户代码，以创建0到n-1之间整数出现频率的直方图。频率保存在一个实例变量数组中。整数实例变量max存储数值出现的最大频率（绘制直方图时用于缩放y轴）。示例客户程序是Bernoulli（程序2.2.6）的一个修改版本，但由于使用了Histogram数据类型，因此实现起来很简单。

393

海龟绘图 在计算中应当将一个大任务清晰地划分成若干个小任务。在面向对象程序设计中，我们拓展该设计理念，被划分的还应该包括任务的数据（或状态）。减少状态的数量对于简化计算过程非常有价值。接下来，我们讨论海龟绘图，其基于数据类型的 API 定义如下：

```
public class Turtle

    Turtle(double x0, double y0, double a0) 在坐标点（x0，y0）处创建一个面向
                                              a0度（x轴逆时针夹角方向）的海龟

    void turnLeft(double delta)              逆时针旋转delta度

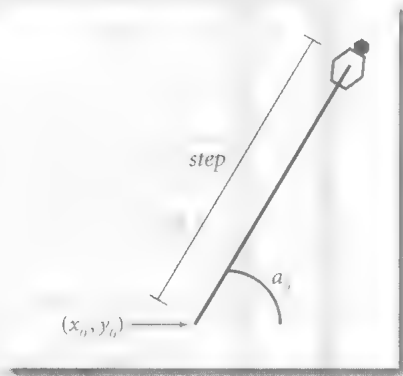
    void goForward(double step)              移动step距离，绘制一条直线
```

海龟绘图 API（见程序 3.2.4）

想象一下，一个生活在单位正方形内的海龟，它在移动时会画出直线。海龟可以沿直线移动指定的距离，也可以向左（逆时针）旋转指定的角度。根据 API，当我们创建一个海龟时，我们把它放在一个指定的位置点上，面对一个指定的方向。然后，通过给海龟提供一系列的 `goForward()` 和 `turnLeft()` 命令来创建绘图。

```
double x0 = 0.5;
double y0 = 0.0;
double a0 = 60.0;
double step = Math.sqrt(3)/2;
Turtle turtle = new Turtle(x0, y0, a0);
turtle.goForward(step);
```

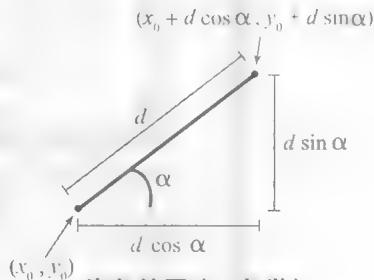
例如，为了绘制一个等边三角形，我们在 $(0.5, 0)$ 处创建一个海龟，面向与原点逆时针成 60° 角，指示它向前移动一步，逆时针旋转 120° ，再向前移动一步，然后逆时针旋转 120° ，再向前迈出第三步，完成三角形的绘制。事实上，我们要研究的所有的 `turtle` 客户都是简单创建一只海龟，然后交错发出一系列移动和旋转的指令，只是移动的步长和旋转的角度各不相同而已。你稍后会发现，这个简单的模型允许我们创建任意复杂的图像，应用十分广泛。



海龟绘图的第一步

`Turtle`（程序 3.2.4）是其 API 的一种实现，程序使用了模块 `StdDraw`。`Turtle` 类维护三个实例变量：海龟位置的坐标和当前的方向（从 x 轴逆时针方向测量）。`Turtle` 类实现了用于更新这些实例变量的两个方法，所以 `Turtle` 类是可变对象。这些更新方法的实现很简单：`turnLeft(delta)` 将当前角度增加 δ ，`goForward(step)` 将当前 x 坐标递增 step 乘以当前角度的余弦值，把当前 y 坐标递增 step 乘以当前角度的正弦值。

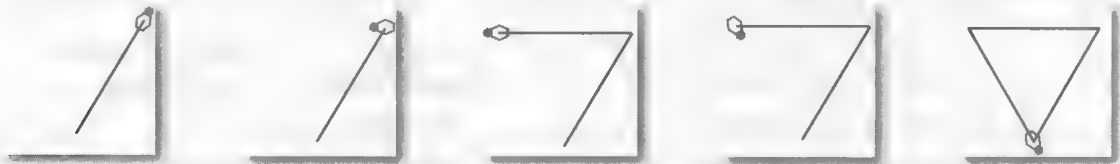
`Turtle` 的测试客户程序接收一个整数作为命令行参数 n ，并绘制一个 n 边正多边形。如果你对初等解析几何感兴趣，可以验证结果的正确性。无论是否进行验证，请你思考一个问题：如何计算多边形中所有点的坐标。海龟绘图方法的简洁性是非常有吸引力的。简单来说，海龟绘图可以作为描述各种几何形状的抽象。例如，通过把 n 设置足够大，可以获得一个近似的圆。



海龟绘图（三角学）

我们像使用任何其他对象一样使用 `Turtle`。程序可以创建 `Turtle` 对象的数组，也可以将其作为参数传递给函数等。这些例子体现了这些功能，证明创建像 `Turtle` 这样的数据类型是非常简单且实用的。针对各种规则的多边形，程序可以计算所有点的坐标并连接直线来获得图形，但是使用 `Turtle` 类会更容易实现。海龟绘图的例子体现了数据抽象的价值。

```
turtle.goForward(step);  turtle.turnLeft(120.0);  turtle.goForward(step);  turtle.turnLeft(120.0);  turtle.goForward(step);
```



你的第一个海龟绘图

程序3.2.4 海龟绘图

```

public class Turtle
{
    private double x, y;
    private double angle;
    public Turtle(double x0, double y0, double a0)
    { x = x0; y = y0; angle = a0; }
    public void turnLeft(double delta)
    { angle += delta; }
    public void goForward(double step)
    { // 计算新位置, 移动并画线
      double oldx = x, oldy = y;
      x += step * Math.cos(Math.toRadians(angle));
      y += step * Math.sin(Math.toRadians(angle));
      StdDraw.line(oldx, oldy, x, y);
    }
    public static void main(String[] args)
    { // 绘制一个正n边形
      int n = Integer.parseInt(args[0]);
      double angle = 360.0 / n;
      double step = Math.sin(Math.toRadians(angle/2));
      Turtle turtle = new Turtle(0.5, 0.0, angle/2);
      for (int i = 0; i < n; i++)
      {
          turtle.goForward(step);
          turtle.turnLeft(angle);
      }
    }
}

```

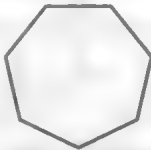
x, y 坐标位置(在单位正方形内)
angle 运动方向(角度, 与x轴逆时针方向夹角)

实现的数据类型支持海龟绘图, 经常用于简单图形的创建

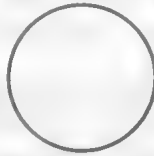
% java Turtle 3



% java Turtle 7



% java Turtle 1000



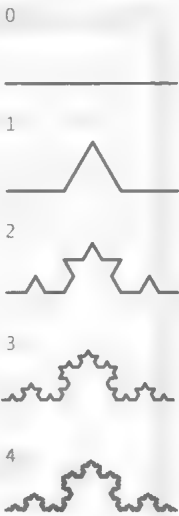
递归图形。0 阶的科赫曲线 (Koch curve) 是一条直线段。通过以下方法, 可以形成一条 n 阶的科赫曲线: 绘制一条 $n-1$ 阶的科赫曲线, 向左旋转 60° , 再绘制一条 $n-1$ 阶的科赫曲线, 向右转 120° (向左转 -120°), 绘制第三条 $n-1$ 阶的科赫曲线, 向左转 60° , 绘制第四条 $n-1$ 阶的科赫曲线。显然, 这些递归指令可以通过海龟绘图客户代码实现。经过适当的修改, 类似的递归方案可以用来模拟自然界中发现的自相似模式, 比如雪花。

客户端代码思路非常直截了当, 除了步长的值有些麻烦。如果仔细阅读前几个示例, 读者将发现 (并能够通过归纳法证明), n 阶曲线的宽度是步长的 3^n 倍, 所以设置步长为 $1/3^n$, 会产生一条宽度为 1 的曲线。类似地, n 阶曲线中的步数是 4^n , 所以如果 n 比较大, Koch 程序将无法正常工作。

我们可以找到很多这样的递归模式的示例, 这些模式已经被不同文化背景的数学家、科学家和艺术家研究和开发过了。在这里我们关注的是, 海龟绘图抽象可以大大简化绘制递归图形的客户代码。

```
public class Koch
{
    public static void koch(int n, double step, Turtle turtle)
    {
        if (n == 0)
        {
            turtle.goForward(step);
            return;
        }
        koch(n-1, step, turtle);
        turtle.turnLeft(60.0);
        koch(n-1, step, turtle);
        turtle.turnLeft(-120.0);
        koch(n-1, step, turtle);
        turtle.turnLeft(60.0);
        koch(n-1, step, turtle);
    }

    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        double step = 1.0 / Math.pow(3.0, n);
        Turtle turtle = new Turtle(0.0, 0.0, 0.0);
        koch(n, step, turtle);
    }
}
```



397

用海龟绘图绘制科赫曲线

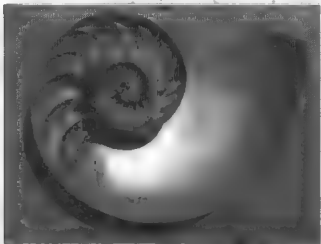
等角螺线。经过 4 个步骤绘制科赫曲线后，也许海龟已经有些疲倦了。因此，假设每次迈步时，海龟的步长都会以微小的常数递减。绘制的图形会发生什么变化？特别地，修改程序 3.2.4 中的多边形绘制的客户程序来回答这个问题，程序会产生一个被称为对数螺旋的图形，这个曲线在自然界的许多环境中都可以找到。

Spiral（程序 3.2.5）是这条曲线的一个实现。程序的命令行参数为 n 和衰变因子，指示海龟交替地前行和转动，直到它缠绕 10 次。从程序中给出的四个例子中可以看出，如果衰变因子大于 1，则螺旋的路径会到达图形的中心。参数 n 控制螺旋的形状。建议读者尝试使用不同的参数测试 Spiral，以理解各参数如何控制螺旋曲线的行为。

对数螺旋最早是由笛卡儿（René Descartes）于 1638 年提出的。雅各布·伯努利（Jacob Bernoulli）对其数学性质感到非常惊讶，将其命名为等角螺线（*spira mirabilis*），甚至要求将其刻在自己的墓碑上。许多人也认为它是“神奇的”，因为这种精确曲线清楚地存在于各种各样的自然现象中。下面是三个例子：一个鹦鹉螺的贝壳，一个螺旋星系的旋臂，一个热带风暴的形成图。科学家还观察到，鹰接近猎物时的路径和带电粒子垂直向均匀磁场移动的曲线都是螺旋曲线。

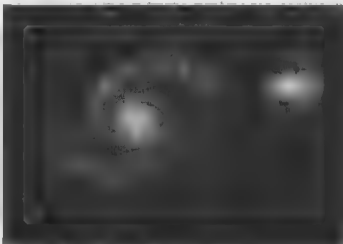
科学探究的目标之一就是为复杂的自然现象提供一个简单但准确的模型。我们不知疲倦的海龟肯定经受住了考验！

鹦鹉螺壳



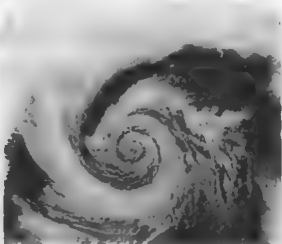
图片来源：Chris 73(CC by-SA license)

螺旋星系



图片来源：NASA and ESA

热带暴风雨



图片来源：NASA

自然界等角螺线示例

398

程序3.2.5 等角螺旋线

```
public class Spiral
{
    public static void main(String[] args)
    {
        int n          = Integer.parseInt(args[0]);
        double decay    = Double.parseDouble(args[1]);
        double angle    = 360.0 / n;
        double step     = Math.sin(Math.toRadians(angle/2));
        Turtle turtle   = new Turtle(0.5, 0, angle/2);

        for (int i = 0; i < 10 * 360 / angle; i++)
        {
            step /= decay;
            turtle.goForward(step);
            turtle.turnLeft(angle);
        }
    }
}
```

step	步长
decay	递减因子
angle	旋转量
turtle	疲倦的海龟

这个代码是程序3.2.4中测试客户程序的修改版本；程序每一步递减步长，并且环绕了大约10次。angle控制形状；递减因子decay控制螺旋的本质。

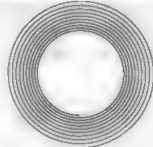
% java Spiral 3 1.0



% java Spiral 3 1.2



% java Spiral 1440 1.00004



% java Spiral 1440 1.0004



布朗运动。假设“海龟”有很多只。因此，请读者想象一只迷失方向的海龟（同样遵循其标准的交替转向和前行规则）在每一步之前随机转向。很容易绘制该海龟经过数百万步后的路径，而这样的路径在自然界许多情况下存在。1827年，植物学家罗伯特·布朗（Robert Brown）通过显微镜观察到，从花粉中喷出的微小颗粒浸入水中后，似乎以迷失方向的海龟的运动方式随机移动。这个过程后来被称为布朗运动，并且启发了阿尔伯特·爱因斯坦（Albert Einstein）对物质原子本质的洞察。

也许海龟有小伙伴，而且数量很多。在它们移动了很长一段时间后，它们的路径合并在一起，成为一个不可分割的路径。现在，天体物理学家使用这个模型来了解遥远星系的观测特性。

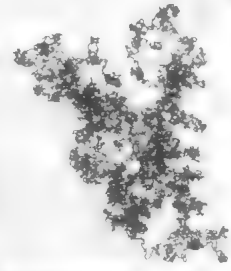
海龟绘图最初由麻省理工学院的西蒙·派珀特（Seymour Papert）在20世纪60年代开发，作为教育程序设计语言Logo的一部分，Logo语言至今仍在“玩具”计算机中使用。然而，正如若干科学实例表明的，海龟绘图不是“玩具”，海龟绘图也有许多商业应用。例如，它是PostScript的基础，PostScript是一种用于创建大多数报纸、杂志和书籍的打印页面的编程语言。在本节内容中，Turtle是一个典型的面向对象程序设计实例，表明了少量的保存状态（使用对象完成数据抽象，而不仅仅是函数）可以大大简化计算过程。

```

public class DrunkenTurtle
{
    public static void main(String[] args)
    {
        int trials = Integer.parseInt(args[0]);
        double step = Double.parseDouble(args[1]);
        Turtle turtle = new Turtle(0.5, 0.5, 0.0);
        for (int t = 0; t < trials; t++)
        {
            turtle.turnLeft(StdRandom.uniform(0.0, 360.0));
            turtle.goForward(step);
        }
    }
}

```

% java DrunkenTurtle 10000 0.01



400 一只醉酒的海龟的布朗运动（随机移动一个固定的距离）

```

public class DrunkenTurtles
{
    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]); // 海龟的数量
        int trials = Integer.parseInt(args[1]); // 步数
        double step = Double.parseDouble(args[2]); // 步长
        Turtle[] turtles = new Turtle[n];
        for (int i = 0; i < n; i++)
        {
            double x = StdRandom.uniform(0.0, 1.0);
            double y = StdRandom.uniform(0.0, 1.0);
            turtles[i] = new Turtle(x, y, 0.0);
        }
        for (int t = 0; t < trials; t++)
        {
            // 所有的海龟走一步
            for (int i = 0; i < n; i++)
            {
                // 海龟i向一个随机的方向走一步
                turtles[i].turnLeft(StdRandom.uniform(0.0, 360.0));
                turtles[i].goForward(step);
            }
        }
    }
}

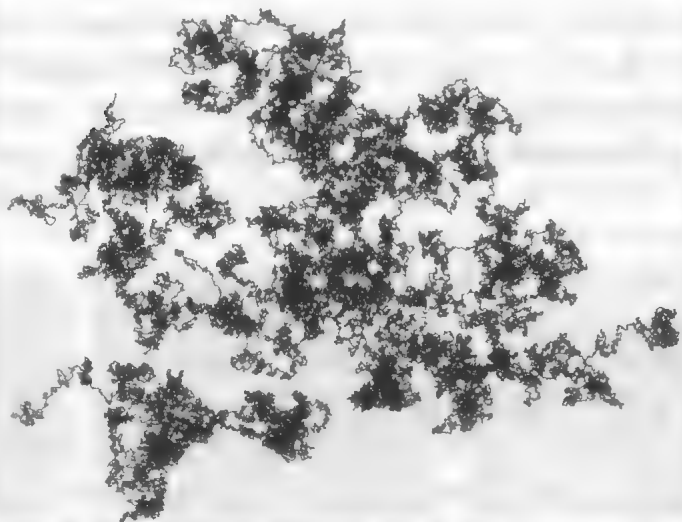
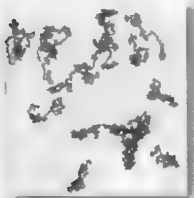
```

20 500 0.005

% java DrunkenTurtles 20 5000 0.005



20 1000 0.005



复数 复数是 $x+iy$ 形式的数字，其中 x 和 y 是实数， i 是 -1 的平方根。 x 被称为复数的实部， y 被称为复数的虚部。这个术语源于这个概念，即 -1 的平方根为虚数，因为没有任何实数对应于这个值。复数是典型的数学抽象：不管人们是否相信“ -1 ”的平方根存在实际意义，复数有助于我们理解自然界。复数被广泛应用于应用数学，在许多科学和工程学科中发挥着重要的作用。复数还被用于建模各种物理系统，从电路到声波到电磁场。这些模型往往需要大量涉及操作复数的计算，复数运算基于规范的数学运算，所以我们要编写计算机程序来实现。简而言之，我们需要一种新的数据类型。

为复数定义数据类型是面向对象程序设计的一个典型例子。没有一种编程语言可以提供给我们可能需要的所有数学抽象的实现，但是自定义数据类型的能力使我们不仅能够编写程序来轻松操纵抽象概念，如复数、多项式、向量和矩阵，而且还能赋予我们利用抽象思维思考问题的自由。

复数运算所需要的基本运算包括：利用交换律、结合律和分配律（以及等式 $i^2=-1$ ）进行的加法、乘法等基础代数计算；计算复数的模数；返回实部和虚部等。这些计算的规则如下：

- 加法： $(x+iy)+(v+iw)=(x+v)+i(y+w)$
- 乘法： $(x+iy) \times (v+iw)=(xv-yw)+i(yv+xw)$
- 模数： $|x+iy|=\sqrt{x^2+y^2}$
- 实部： $\text{Re}(x+iy)=x$
- 虚部： $\text{Im}(x+iy)=y$

例如，如果 $a=3+4i$ ， $b=2+3i$ ，那么 $a+b=5+7i$ ， $a \times b=18+i$ ， $\text{Re}(a)=3$ ， $\text{Im}(a)=4$ ， $|a|=5$ 。

基于上述基本定义，实现复数数据类型的方法变得十分清楚明了。像往常一样，我们从定义数据类型运算操作的 API 开始：

402

```
public class Complex
{
    Complex(double real, double imag)
    Complex plus(Complex b)
    Complex times(Complex b)
    double abs()
    double re()
    double im()
    String toString()
}
```

这个数和b的和
这个数和b的乘积
求模
实部
虚部
字符串表示

复数的 API (参见程序 3.2.6)

为了简单起见，在本书正文中，我们只关注 API 中的基本运算，在练习 3.2.19 则要求读者考虑 API 中的其他一些有用的操作。

Complex（程序 3.2.6）是实现这个 API 的一个类。它具有与 Charge（以及每个 Java 数据类型实现）相同的组件：实例变量（`re` 和 `im`）、构造函数、实例方法（`plus()`、`times()`、`abs()`、`re()`、`im()` 和 `toString()`）和一个测试客户程序。测试客户程序首先将设置 z_0 为 $1+i$ ，然后设置 z 为 z_0 ，然后计算如下表达式：

$$z=z^2+z_0=(1+i)^2+(1+i)=(1+2i-1)+(1+i)=1+3i$$

$$z = z^2 + z_0 = (1+3i)^2 + (1+i) = (1+6i-9) + (1+i) = -7+7i$$

程序代码很简单，与本章前述代码类似，只有一个不同之处：实现算术方法的代码使用了一种新的机制来访问对象值。

访问相同类型的其他对象的实例变量。实例方法 `plus()` 和 `times()` 的实现需要访问两个对象中的值：作为参数传递的对象和调用方法的对象。当客户端调用 `a.plus(b)` 时，可以像往常一样使用名称 `re` 和 `im` 来访问对象 `a` 的实例变量。为了访问 `b` 的实例变量，可以使用代码 `b.re` 和 `b.im`。将实例变量声明为 `private` 意味着不能直接从另一个类访问实例变量。但是，在同一个类中，可以直接访问任何对象的实例变量，而不仅仅是调用者的实例变量。

403

创建并返回新的对象。观察 `plus()` 和 `times()` 向客户端提供返回值的方式就会发现：它们需要返回一个新的 `Complex` 值，因此，它们各自计算所需的实部和虚部，并使用实部和虚部创建一个新对象，然后返回该对象的引用。通过操作 `Complex` 类型的局部变量，这种设计使得客户程序可以按照更自然的方式处理复数。

方法调用链。在 `main()` 方法中，我们将两个方法调用成一个紧凑的 Java 表达式 `z.times(z).plus(z0)`，对应于数学表达式 $z^2 + z_0$ 。这种用法很方便，因为我们不必为中间值创建变量名称。也就是说，我们可以使用任何对象引用来调用一个方法，即使没有名称（如子表达式的计算结果）。研究该表达式就会发现，这种方式没有歧义：方法的调用从左向右移动，每个方法都会返回对 `Complex` 对象的引用，该对象用于调用链中的下一个实例方法。如果需要，我们可以使用圆括号来覆盖默认的优先顺序（例如，Java 表达式 `z.times(z.plus(z0))`）对应于数学表达式 $z(z+z_0)$ 。

Final 类型实例变量。`Complex` 中的两个实例变量是 `final` 类型的，这意味着每个 `Complex` 对象的值是在创建时设置的，并且在对象的整个生命周期中不会变化。我们在 3.3 节讨论这个设计的原因。

复数是应用数学中复杂计算的基础，应用十分广泛。通过 `Complex`，我们可以专注于开发使用复数的应用程序，而不用担心重新实现诸如 `times()`、`abs()` 等方法。这种方法实现一次，就可以重复使用，而不需要将这些代码复制到任何使用复数的应用程序中。这种方法不仅可以节省调试时间，还可以根据需要对实现方式进行更改或改进，因为方法与客户程序是分开的。在计算中应当清晰地分离数据和相关的计算任务。为了体验复数计算的本质和复数抽象的应用，接下来我们将讨论一个著名的 `Complex` 客户程序示例。

404

曼德布洛特集合 曼德布洛特集合 (Mandelbrot set) 是由本华·曼德布洛特 (Benoit Mandelbrot) 发现的一组特定的复数。这个集合有许多有趣的属性。它是一个分形图案，与本书中我们已经看到的巴恩斯利蕨、谢尔宾斯基三角形、布朗桥、科赫曲线、醉酒的海龟和其他递归（自相似）模式和程序有关。这种模式可以在各种自然现象中找到，这些模型和程序在现代科学中有着重要的作用。

曼德布洛特集合中的点集不能用简单的数学方程来描述，只能由一种算法定义，因此，非常适合于实现为复数类的客户程序：我们通过编写一个绘制图形的程序来研究曼德布洛特集合。

确定一个复数 z_0 是否属于曼德布洛特集合的规则很简单。假设有复数序列 $z_0, z_1, z_2, \dots, z_i, \dots$ ，其中如果 $z_{i+1} = (z_i)^2 + z_0$ 。例如，下面表格列举了当 $z_0 = 1+i$ 时，序列中的前几项元素：

程序3.2.6 复数

```
public class Complex
{
    private final double re;
    private final double im;

    public Complex(double real, double imag)
    { re = real; im = imag; }

    public Complex plus(Complex b)
    { // 返回这个数字和b的和
      double real = re + b.re;
      double imag = im + b.im;
      return new Complex(real, imag);
    }

    public Complex times(Complex b)
    { // 返回这个数字和b的乘积
      double real = re * b.re - im * b.im;
      double imag = re * b.im + im * b.re;
      return new Complex(real, imag);
    }

    public double abs()
    { return Math.sqrt(re*re + im*im); }

    public double re() { return re; }
    public double im() { return im; }

    public String toString()
    { return re + " + " + im + "i"; }

    public static void main(String[] args)
    {
        Complex z0 = new Complex(1.0, 1.0);
        Complex z = z0;
        z = z.times(z).plus(z0);
        z = z.times(z).plus(z0);
        StdOut.println(z);
    }
}
```

re 实部
im 虚部

这种数据类型是编写处理复数的Java程序的基础

```
% java Complex
-7.0 + 7.0i
```

i	z_i	$(z_i)^2$	$(z_i)^2 + z_0$	z_{i+1}
0	1 + i	1 + 2i + i ² = 2i	-2i + (1 + i) =	1 + 3i
1	1 + 3i	1 + 6i + 9i ² = -8 + 6i	-8 + 6i + (1 + i) =	-7 + 7i
2	-7 + 7i	49 - 98i + 49i ² = -98i	-98i + (1 + i) =	1 - 97i

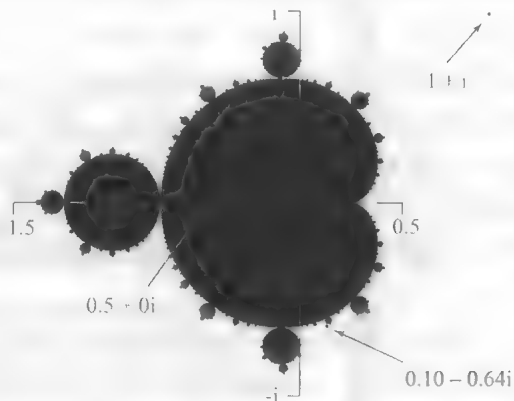
曼德布洛特序列计算

现在，如果序列 $|z_i|$ 发散到无穷大，则 z_0 不属于曼德布洛特集合。如果序列有边界，则 z_0 属于曼德布洛特集合。对许多复数值执行这个测试很简单，但在很多情况下测试需要大量的计算，如下表所示：

z_0	0 + 0i	2 + 0i	1 + i	0 + i	-0.5 + 0i	-0.10 - 0.64i
z_1	0	6	1 + 3i	-1 + i	-0.25	-0.30 - 0.77i
z_2	0	38	-7 + 7i	-i	-0.44	-0.40 - 0.18i
z_3	0	1446	1 - 97i	-1 + i	-0.31	0.23 - 0.50i
z_4	0	2090918	-9407 - 193i	-i	-0.40	-0.09 - 0.87i
⋮	⋮	⋮	⋮	⋮	⋮	⋮
in set?	yes	no	no	yes	yes	yes

曼德布洛特序列的前几项

为了简单起见，上表最右边两列数据仅保留两位小数的精度。在某些情况下，我们可以证明数值是否属于曼德布洛特集合。例如， $0+0i$ 当然在集合中（因为其序列中所有数的模数都是 0），而 $2+0i$ 显然不在集合中（因为序列中数的模数为前一个的平方加 2，最终会增长到无限大）。在有些情况下，增长的趋势也是显而易见的。例如， $1+i$ 似乎也不在集合中。也有的序列表现出周期性行为特征。例如， $0+i$ 的序列在 $-1+i$ 和 $-i$ 之间交替变化。有些序列会持续很长时间后，数值的模数才会开始变大。



曼德布洛特集合

为了可视化曼德布洛特集合，我们对复数点进行采样，与绘制实数函数时使用的采样方法一致。每个复数 $x+iy$ 对应于平面上的一个坐标点 (x, y) ，所以可以按照以下方法绘制结果：对于一个给定的分辨率 n ，我们将给定的正方形区域划分为一个均匀分布的 $n \times n$ 像素网格，并且如果对应点属于曼德布洛特集合，则把对应的像素绘制成黑色，否则就绘制成白色。绘制结果是一个神奇的图案，所有的黑色点连接在一起，大致落在以点 $-1/2+0i$ 为中心的 2×2 正方形之内。 n 值越大，产生的图像分辨率越高，但代价是需要更多的计算量。通过观察，可以发现所绘图像的自相似性。例如，在主要的黑色心形区域的轮廓周围，出现自相似的相同的球状图案作为附属，其大小与程序 1.2.1 的简单标尺功能相似。当我们放大心形边缘附近时，会出现微小的自相似心形图！

但是，究竟是如何产生这种情况的呢？事实上，没有人知道准确答案，因为没有简单的测试可以让我们得出这样一个结论：一个点确实在集合中。给定一个复数点，我们可以计算序列的前几项，但可能无法确定序列是否有边界。存在一个简单的数学测试，可以用于判断一个复数不在集合中：如果序列中存在任何一个数的模超过 2（如 $1+3i$ ），那么序列肯定发散。

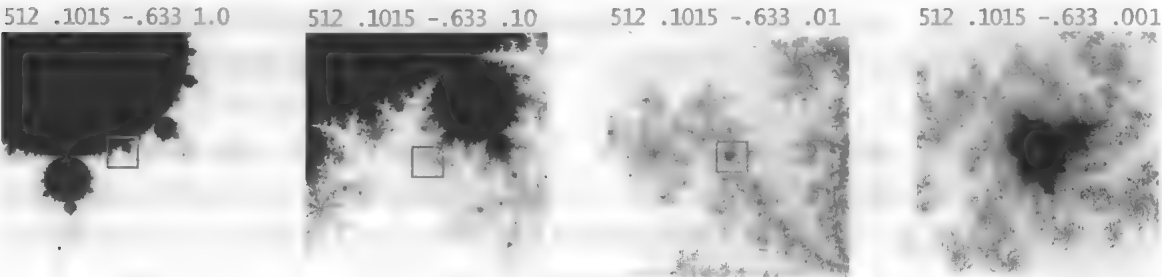
[407]

Mandelbrot（程序 3.2.7）使用这个测试来绘制曼德布洛特集合的可视化图形。由于集合的点不能仅仅使用黑或白两种颜色表示，因此我们在可视化表示中使用了灰度。计算曼德布洛特集合规则的函数是 `mand()`，该函数的参数是复数 z_0 和 `int` 型的 `max`，从 z_0 开始计算曼德布洛特迭代序列，返回模数保持小于（或等于）2 的迭代次数，直到极限 `max`。

对于每个像素，Mandelbrot 中的 `main()` 方法计算与该像素对应的复数 z_0 ，然后通过计算 `255-mand(z0,255)` 为该像素创建灰度颜色值。任何一个像素都对应于一个复数，如果这个复数不在曼德布洛特集合中，那么它不是黑色的，因为它的序列中数字的模数超过了 2（因此会发散到无穷大）。黑色像素点（灰度值 0）对应于我们假定属于集合的点，因为在首个 255 次的曼德布洛特迭代期间，其模数不超过 2。

这个简单的程序产生的图像极其复杂，即使放大平面图像的一部分也是如此。如果要获得更奇妙的显示效果，我们可以绘制一个彩色的图（见练习 3.2.35）。曼德布洛特集合仅通过迭代一个简单的函数 $f(z)=(z^2+z_0)$ 而来，我们也可以研究其他这样的函数，获取更丰富的效果和特性。

代码的简单性掩盖了计算的复杂性。在 512×512 大小的图像中约有 25 万像素，而所有黑色像素点需要 255 次曼德布洛特迭代，所以用 Mandelbrot 生成一幅图像，需要对 `Complex` 值进行数以百万计的操作。



集合的放大

虽然展示的效果很炫，但我们对 Mandelbrot 的主要兴趣在于它是利用 Complex 实现的一个示例客户程序，用来说明使用未内置在 Java 中的数据类型（复数）进行计算也是一种自然且实用的编程活动。得益于 Complex 的设计和实现，Mandelbrot 中的客户代码是一个简单而自然的计算表达式。你也可以在不使用 Complex 的情况下实现 Mandelbrot，但是代码本质上必须将程序 3.2.6 和程序 3.2.7 中的代码合并在一起，这将更加难以理解。正应了我们前面说的：在计算中应当清晰地分离数据和相关的计算任务。

408

程序3.2.7 曼德布洛特集合

```
import java.awt.Color;
public class Mandelbrot
{
    private static int mand(Complex z0, int max)
    {
        Complex z = z0;
        for (int t = 0; t < max; t++)
        {
            if (z.abs() > 2.0) return t;
            z = z.times(z).plus(z0);
        }
        return max;
    }

    public static void main(String[] args)
    {
        double xc = Double.parseDouble(args[0]);
        double yc = Double.parseDouble(args[1]);
        double size = Double.parseDouble(args[2]);
        int n = 512;
        Picture picture = new Picture(n, n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
            {
                double x0 = xc - size/2 + size*i/n;
                double y0 = yc - size/2 + size*j/n;
                Complex z0 = new Complex(x0, y0);
                int gray = 255 - mand(z0, 255);
                Color c = new Color(gray, gray, gray);
                picture.set(i, n-1-j, c);
            }
        picture.show();
    }
}
```

x0, y0	正方形中的点
z0	$x_0 + iy_0$
max	迭代次数上限
xc, yc	正方形的中心点
size	正方形的规定尺寸
n	网格是 $n \times n$ 像素
pic	输出的图像
c	输出的像素颜色

-5 0 2

.1015 -.633 .01

该程序接收三个命令行参数来指定正方形区域的中心和大小，并以数字图像的形式，显示在均匀分布像素的区域中曼德布洛特集合的采样结果。通过计算相应复数的曼德布洛特序列的迭代次数，确定每个像素的灰度值，最大值为255。

409

商业数据处理 面向对象程序设计的开发推动力之一是大量用于商业数据处理的可靠软件的需求。作为说明，我们接下来将讨论一个金融机构可能用来跟踪客户信息的数据类型的例子。

假设证券经纪人需要维护客户账户信息，包括各种股票的份额。也就是说，证券经纪人需要处理的一系列数据包括：客户名称、持有的不同股票数量、每只股票的代码和份额以及手头的现金数量。为了处理一个账户信息，证券经纪人至少需要定义如下 API 对应的运算操作：

public class StockAccount	
StockAccount(String filename)	在文件中创建一个新的账户
double valueOf()	账户总额
void buy(int amount, String symbol)	将股票的股份添加到账户
void sell(int amount, String symbol)	从账户中减去股票的份额
void save(String filename)	保存账户到文件
void printReport()	打印股票和价值的详细报告

处理股票账户的 API (见程序 3.2.8)

很显然，证券经纪人需要买入股票、卖出股票和向客户提供报表，但理解此类数据处理的首要任务是考虑 API 中的 `StockAccount()` 构造函数和 `save()` 方法。客户信息必须长久保存，所以需要保存在文件或数据库中。为了处理一个账户，客户端程序需要从相应的文件中读取信息：根据需求处理信息；并且如果信息发生更改，则将信息写回文件，并保存文字以备后用。为了实现此类处理，我们需要一个用于存储账户信息的文件格式和一个在内存中的内部表述方式（或一个数据结构）。

作为一个（异想天开的）例子，我们想象证券经纪人正在管理计算之父艾伦·图灵（Alan Turing）的软件公司的部分股票投资份额。顺便说一句：图灵的生平故事是一个值得进一步了解的有趣的故事。比如他从事过计算密码学的研究，大大加速了第二次世界大战的结束；他为现代理论计算机科学发展奠定了基础，设计和制造了第一台计算机；他还是人工智能研究的先驱者。也许可以假设，在 20 世纪中叶，无论图灵作为一名学术人员的财务状况如何，只要他投资小部分资产，就可以足够乐观地估计其对当代计算机软件的潜在巨大影响。

410

文件格式。现代系统通常使用文本文件（即使是数据）以最小化程序对格式的依赖。为了简单起见，我们使用直接的表示方法，罗列账户持有人的名字（一个字符串）、现金余额（一个浮点数）和持有的股票数量（一个整数），随后每个股票信息占一行（包括股票份额和股票代码），如右图所示。使用如 `<Name>`、`<Number of shares>` 等标签标记所有数据其实是个明智之举，这样可以进一步减少程序对数据的依赖。但为了简洁起见，示例中忽略了这些标签。

```
% more Turing.txt
Turing, Alan
10.24
4
100 ADBE
25 GOOG
97 IBM
250 MSFT

文件格式
```

数据结构。为了表示 Java 程序处理的信息，我们使用实例变量。实例变量指定了信息的类型，并提供了代码中需要引用的结构。例如，为了实现我们的例子需要以下实例变量：

- 一个表示账户名称的字符串值。
- 一个表示现金余额的浮点数值。
- 一个表示股票数量的整数值。
- 一个表示股票代码的字符串数组。
- 一个表示股票份额的整数数组。

我们在 StockAccount (程序 3.2.8) 的实例变量声明部分实现了这些变量的设计。数组 stocks[] 和 shares[] 被称为平行数组。对于给定的索引 i, stocks[i] 表示股票代码, shares[i] 表示账户中该股票的持有份额。一种替代的设计是, 为股票定义一个单独的数据类型, 在 StockAccount 中创建该数据类型的对象数组以操作每种股票的信息。

StockAccount 包含一个构造函数, 用于以指定的格式读取一个文件, 并用这些信息创建一个账户。此外, 经纪人需要向客户提供周期性的详细报表, 经纪人可使用 StockAccount 中的 printReport() 代码, 依靠 StockQuote (程序 3.1.8) 从 Web 上检索每个股票的价格。

```
public class StockAccount
{
    private final String name;
    private double cash;
    private int n;
    private int[] shares;
    private String[] stocks;
    ...
}
```

411

数据结构蓝图

```
public void printReport()
{
    StdOut.println(name);
    double total = cash;
    for (int i = 0; i < n; i++)
    {
        int amount = shares[i];
        double price = StockQuote.priceOf(stocks[i]);
        total += amount * price;
        StdOut.printf("%4d %5s ", amount, stocks[i]);
        StdOut.printf("%9.2f %11.2f\n", price, amount*price);
    }
    StdOut.printf("%21s %10.2f\n", "Cash: ", cash);
    StdOut.printf("%21s %10.2f\n", "Total:", total);
}
```

valueOf() 和 save() 函数的实现很简单 (参见练习 3.2.22)。buy() 和 sell() 的实现需要使用 4.4 节介绍的基本机制, 所以我们将其推迟到练习 4.4.65 中再做讨论。

一方面, 这个客户程序展示了一种计算模式, 该模式是 20 世纪 50 年代计算演变的主要推动力之一。银行和其他公司正是因为需要做这样的财务报告而购买了早期的计算机。例如, 格式化的输出就是为这样的应用程序而开发的。另一方面, 这个客户程序是现代的以网络为中心计算的一个示例, 因为其直接从网上获取信息, 而不通过 Web 浏览器。

除了这些基本的方法之外, 实现这些设计理念的实际应用可能会使用其他许多客户程序。例如, 证券经纪人可能希望为所有账户创建一个数组, 然后修改这些账户信息, 并通过网站实际提交这些交易。当然, 实现这种功能的代码需要非常谨慎!

412

当你在第 2 章中学习如何定义在一个程序 (或其他程序) 的多个地方使用的函数后, 你就已从单个文件中简单语句序列的程序世界进入模块化编程的世界, 总结成我们的程序设计理念就是: 在计算中应当尽可能清晰地将计算任务划分成相互独立的子任务。与之相比, 本章介绍的数据类型引导读者从少量内置数据类型的世界进入一个可以自定义数据类型的世界。这种新能力意义深远, 极大地扩展了读者的编程范围和能力。就像函数的概念一样, 一旦你学会了实现和使用数据类型, 你会发现不使用自定义数据类型的程序十分粗糙和原始。

程序3.2.8 股票账户

```
public class StockAccount
{
    private final String name;
    private double cash;
    private int n;
    private int[] shares;
    private String[] stocks;

    public StockAccount(String filename)
    { // 从指定的文件中构建数据结构
      In in = new In(filename);
      name = in.readLine();
      cash = in.readDouble();
      n = in.readInt();
      shares = new int[n];
      stocks = new String[n];
      for (int i = 0; i < n; i++)
      { // 处理股票
        shares[i] = in.readInt();
        stocks[i] = in.readString();
      }
    }

    public static void main(String[] args)
    {
      StockAccount account = new StockAccount(args[0]);
      account.printReport();
    }
}
```

name	顾客姓名
cash	现金余额
n	股票数量
shares[]	份额计数
stocks[]	股票的符号

这个类用于处理股票账户。这个类说明了用于商业数据处理的面向对象编程的典型用法。有关priceOf()、save()、buy()和sell()的实现在练习3.2.22和4.4.65中详细讨论，关于printReport()请阅读本节中的相关内容。

```
% more Turing.txt
Turing, Alan
10.24
4
100 ADBE
25 GOOG
97 IBM
250 MSFT
```

```
% java StockAccount Turing.txt
Turing, Alan
100 ADBE      70.56      7056.00
25  GOOG     502.30     12557.50
97  IBM      156.54     15184.38
250 MSFT     45.68     11420.00
Cash:        10.24
Total:      46228.12
```

但是，面向对象程序设计不止于结构化数据。面向对象程序设计可以把数据和相关子任务通过运算操作相关联，并将两者分开保存在独立的模块中。通过面向对象的程序设计，我们的设计理念是这样的：在计算中应当清晰地分离数据和相关的计算任务。

我们考虑过的例子就是有说服力的证据，说明面向对象的程序设计方法可以广泛应用于编程活动。无论是设计和构建物理工件、开发软件系统、理解自然界，还是处理信息，首要步骤都是定义适当的抽象，如物理对象的几何描述、软件系统的模块化设计、自然世界的数学模型以及信息的数据结构。当我们想编写程序来操作定义好的抽象的实例时，我们可以把数据抽象实现为 Java 类中的一个数据类型，然后编写 Java 程序来创建和操作这种类型的对象。

每次我们开发一个类，并使用其他类创建和操作由该类定义的数据类型的对象时，我们就是在更高抽象层面上编写程序。在下一节中，我们将讨论这类编程中固有的一些设计挑战。

问答环节

问：实例变量是否具有默认初始值？

413
}
414

答：是的。对于数字类型，它们自动设置为 0，布尔类型为 false，所有引用类型都为 null。这些值与 Java 自动初始化数组元素的方式一致。这种自动初始化确保每个实例变量总是保存一个有效的（但不一定有意义的）值。编写代码时依赖于初始化值是有争议的：一些有经验的程序员支持这个想法，因为代码可能会更加紧凑（省去了程序员手动填写 0、false、null 等初值的工作——译者注）；还有一部分人则主张避免这种情况，因为对于不了解初始化规则的人来说代码是不透明的。

问：什么是 null？

答：这是一个字符常量，表示没有对象。使用 null 引用来调用实例方法是没有意义的，并导致 NullPointerException 异常。通常情况下，这是一个标志，表示用户未能正确初始化一个对象的实例变量或数组元素。

问：当我声明实例变量时，是否可以将其初始化为默认值以外的值？

答：如果需要，你可以在构造函数中将实例变量初始化为非默认值。与局部变量的内联初始化的规定类似，可以在声明实例变量时为其指定初始值。此内联初始化在构造函数被调用之前发生。

问：每个类都必须有一个构造函数吗？

答：是的，但是如果不指定构造函数，Java 将自动提供一个默认的（没有参数）构造函数。当客户端使用 new 调用这个构造函数时，实例变量像往常一样自动初始化。如果用户指定了一个构造函数，那么默认的无参数构造函数就会自动失效。

问：假设不引入 toString() 方法，而尝试使用 StdOut.println() 打印该类型的对象，会发生什么情况？

答：这时，打印输出的结果是一个对用户不太有用处的整数。

415

问：可以在一个数据类型中定义静态方法吗？

答：当然可以。例如，所有的类都有 main() 方法。但是当静态方法和实例方法混在同一个类中时，很容易混淆。例如，对于涉及多个对象的操作，而在这些操作中，没有一个对象能够很自然地作为发起者调用该方法，那么就会考虑使用静态方法。例如，我们写 `z.abs()` 来得到 `|z|` 非常自然，但是写 `a.plus(b)` 来进行一次 `a+b` 的操作也许并不那么自然。为什么不用 `b.plus(a)`？另一种方法是在 Complex 中定义如下静态方法：

```
public static Complex plus(Complex a, Complex b)
{
    return new Complex(a.re + b.re, a.im + b.im);
}
```

我们通常会避免这种用法，并且使用不会混用静态方法和实例方法的表达式。例如我们通常会避免编写如下代码：

```
z = Complex.plus(Complex.times(z, z), z0)
```

相反，我们会写：

```
z = z.times(z).plus(z0)
```

问：使用 plus() 和 times() 的计算看起来相当烦琐。有什么方法可以在涉及对象（例如 Complex 和 Vector）的表达式中使用像 “+” 和 “*” 这样的符号，以便可以编写更紧凑的表达式，如 `z=z * z+z0`。

答：一些语言（特别是 C++ 和 Python）支持这种功能。这种实现方式被称为运算符重载，但 Java 不支持。像之前的一些情况一样，这是由语言设计者决定的，但是许多 Java 程序员并不认为这是一个很大的损失。运算符重载仅适用于表示数值或代数抽象的类型，占总数的很小一部分，而许多程序调用具有描述性名称的方法（如 `plus()` 和 `times()`）则更便于理解。20 世纪 70 年代的 APL 编程语言，通过坚持每一个操作都用一个符号（包括希腊字母）来表示的方法，把这个问题推到了另一个极端，十分难以理解。

问：在类中，除了参数变量、局部变量和实例变量之外，是否还存在其他类型的变量？

答：如果将 `static` 关键字包含在变量声明中（不在任何方法中），则会创建一个完全不同类型的变量，称为静态变量或类变量。像实例变量一样，静态变量可以被类中的每个方法访问，但是，它们不与任何对象关联——每个类只有一个变量副本。在较老的编程语言中，这样的变量由于其作用范围是整个程序而被称为全局变量。在现代编程中，我们往往注意限制变量的范围，所以很少使用这样的变量。

问：Mandelbrot 创建了数以亿计的 `Complex` 对象。请问这种创建对象的开销是否会使程序变慢？

答：是的，但不会缓慢到无法生成绘图。我们的目标是通过复杂的数字抽象来使得我们的程序易读、易维护。通过复数抽象限制范围可以帮助我们实现上述目标。当然，可以不通过复杂的数字抽象或使用 `Complex` 的不同实现来加速 Mandelbrot。

练习

3.2.1 阅读以下矩形（与坐标轴平行）的数据类型的实现代码，其使用中心点的坐标位置、宽度和高度来表示每个矩形：

```
public class Rectangle
{
    private final double x, y;    // 矩形的中心
    private final double width;   // 矩形的宽
    private final double height;  // 矩形的高

    public Rectangle(double x0, double y0, double w, double h)
    {
        x = x0;
        y = y0;
        width = w;
        height = h;
    }

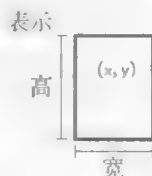
    public double area()
    { return width * height; }

    public double perimeter()
    { /*计算周长*/ }

    public boolean intersects(Rectangle b)
    { /*这个矩形是否和b相交? */ }

    public boolean contains(Rectangle b)
    { /*b在这个矩形里面吗? */ }

    public void draw(Rectangle b)
    { /*在标准图形上绘制矩形。*/ }
}
```



请编写该类的 API，完成该类的实现代码，包括 `perimeter()`、`intersects()` 和 `contains()`。注意：如果两个矩形有一个或多个公共点（不正确的交点），将视为相交。因而，`a.intersects(a)` 和

a.contains(a) 都为 True。

- 3.2.2 请编写一个 Rectangle 测试客户程序，它需要三个命令行参数 n、min 和 max。生成 n 个随机矩形，其宽度和高度均匀分布在 min 和 max 之间。在标准绘图窗口绘制这些矩形，在标准输出中输出其平均面积和周长。
- 3.2.3 将前面的练习代码添加到测试客户程序，以计算彼此相交或包含的矩形对的平均数量。
- 3.2.4 请编写练习 3.2.1 中 Rectangle API 的一种实例，使用矩形的左下角和右上角的 x 和 y 坐标表示矩形。保持 API 不变。
- 3.2.5 以下代码有什么错误？

```
public class Charge
{
    private double rx, ry;    // 位置
    private double q;        // charge值
    public Charge(double x0, double y0, double q0)
    {
        double rx = x0;
        double ry = y0;
        double q = q0;
    }
    ...
}
```

答案：构造函数中的赋值语句也是声明，它们会创建新的局部变量 rx、ry 和 q，这些变量在构造函数完成时超出了范围（不可用）。实例变量 rx、ry 和 q 仍然是其默认值 0。注意：与实例变量名称相同的局部变量被称为映射实例变量。我们将在下一节讨论引用映射实例变量，初学者最好避免这种情况。

- 3.2.6 创建一个数据类型 Location，使用纬度和经度代表地球上一个位置。实现方法 distanceTo()，用来使用大圆距离来计算球上两点间的距离（请参见练习 1.2.33）。
- 3.2.7 实现一个数据类型 Rational，支持加减乘除操作。

```
public class Rational
{
    Rational(int numerator, int denominator)
    Rational plus(Rational b)           这个数字和b的和
    Rational minus(Rational b)          这个数字和b的差
    Rational times(Rational b)          这个数字和b的乘积
    Rational divides(Rational b)        这个数字和b的商
    String toString()                   字符串表示
}
```

使用 Euclid.gcd()（程序 2.3.1）来确保分子和分母没有公因子。请包含一个可以执行所有方法的测试客户程序。不用考虑整数溢出的情况（参见练习 3.3.17）。

- 3.2.8 编写一个数据类型 Interval 实现以下 API：

```
public class Interval
{
    Interval(double left, double right)
    boolean contains(double x)          x是否在这个区间中？
    boolean intersects(Interval b)      这个区间和b是否相交？
    String toString()                   字符串表示
}
```

一个区间被定义为一条线段上大于或等于 `left` 但小于或等于 `right` 的所有点的集合，特别要说明的是，当 `right` 小于 `left` 时，区间为空集。编写一个客户程序（作为过滤器），从命令行接收一个浮点数 `x` 作为命令行参数，在标准输入上读入一系列区间，每个区间由一对 `double` 值表示，输出所有包含 `x` 的区间。

3.2.9 为前一个练习中的 `Interval` 类编写一个客户程序，这次从命令行读入整数型参数 `n`，从标准输入中读取 `n` 个区间（每个区间由一对 `double` 值定义），并打印所有相交的区间对。

420

3.2.10 针对练习 3.2.1，使用 `Interval` 数据类型开发一个 `Rectangle` 的 API 实现，使代码的表达更加简化和清晰。

3.2.11 编写一个数据类型 `Point`，实现以下 API：

```
public class Point
{
    Point(double x, double y)
    double distanceTo(Point q)    此点与q之间的欧氏距离
    String toString()            字符串表示
}
```

3.2.12 给 `Stopwatch` 添加新的方法，允许客户程序停止并重新启动秒表。

3.2.13 使用 `Stopwatch` 比较两种计算谐波数的方法的时间开销：使用 `for` 循环结构（见程序 1.3.5）的方法与 2.3 节中给出的递归方法。

3.2.14 请利用 `Draw` 开发一个新的 `Histogram` 程序，客户程序可以创建多个直方图。在显示图上，在采样均值的位置添加一条红色的垂直线，在显示距离样本均值有两个标准偏差距离处添加一条蓝色垂直线。同时实现一个测试客户程序，用于为翻转硬币（伯努利试验）创建直方图，用偏转概率为 p 的偏向硬币， $p=0.2$ 、 0.4 、 0.6 和 0.8 ，从命令行中获得翻转次数和试验次数，如程序 3.2.3 所示。

3.2.15 修改 `Turtle` 中的测试客户程序，把一个奇数 n 作为命令行参数，用 n 个点绘制一个星形。

3.2.16 修改 `Complex`（程序 3.2.6）中的 `toString()` 方法，使用传统格式输出复数。例如；复数 $3-i$ 应该输出为 $3-i$ 而不是 $3.0+-1.0i$ ；复数 $3i$ 输出为 $3i$ ，而不是 $0.0+3.0i$ 。

3.2.17 编写一个 `Complex` 客户程序，它以三个浮点数 a 、 b 和 c 作为命令行参数，并打印 ax^2+bx+c 的两个（复数）根。

421

3.2.18 编写一个 `Complex` 客户程序 `Roots`，程序在命令行中接收两个浮点数 a 、 b 和一个整数 n ，并打印出 $a+bi$ 的 n 次方根。注意：如果读者不熟悉复数的根的求值操作，请跳过本练习。

3.2.19 实现下列 `Complex` 的 API 扩展：

```
double theta()                这个数字的相位（角度）
Complex minus(Complex b)      这个数字和b的差
Complex conjugate()           这个数字的共轭
Complex divides(Complex b)    把这个数字除以b的结果
Complex power(int b)          这个数字的b次幂
```

写一个测试客户程序，测试所有方法。

3.2.20 假设你想给 `Complex` 类增加一个构造函数，其参数为 `double` 型，并创建一个以该值作为实部（而不管虚部）的复数。编写以下代码：

```
public void Complex(double real)
{
```

```
re = real;
im = 0.0;
}
```

但是，接下来 `Complex c=new Complex(1.0)` 的声明语句编译不通过。为什么？

答案：构造函数不具有返回类型，甚至 `void` 都不可以加。这段代码定义了一个名为 `Complex` 的方法，而不是一个构造函数。应该删除关键字 `void`。

3.2.21 找到一个 `Complex` 型的值，使得 `mand()` 函数的返回值大于 100，然后放大该值，如本书中的示例。

3.2.22 实现 `StockAccount`（程序 3.2.8）的 `valueOf()` 和 `save()` 方法。

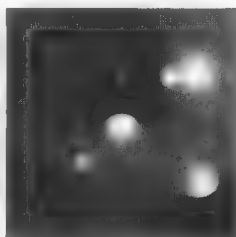
422

创新练习

3.2.23 电势可视化。编写一个程序 `Potential`，根据标准输入（每个带电粒子由其 x 坐标、 y 坐标和电荷值指定）创建一个带电粒子数组，并产生单位正方形中电势的可视化效果。对单位正方形中的点进行采样，对于每个采样点，计算该点的电势（通过对每个带电粒子产生的电位进行求和），并将相应点绘制成与电势成比例的灰色阴影。

% more charges.txt

```
9
.51 .63 -100
.50 .50 40
.50 .72 10
.33 .33 5
.20 .20 -10
.70 .70 10
.82 .72 20
.85 .23 30
.90 .12 -50
```



3.2.24 可变电荷。修改 `Charge`（程序 3.2.1），使电荷值 q 可变，并且添加一个方法 `increaseCharge()`，它接受一个浮点型的参数，并把给定的值添加到 q 上。然后编写一个客户程序，初始化数组：

% java Potential < charges.txt

一组电荷值的电势可视化

```
Charge[] a = new Charge[3];
a[0] = new Charge(0.4, 0.6, 50);
a[1] = new Charge(0.5, 0.5, -5);
a[2] = new Charge(0.6, 0.6, 50);
```

接下来，显示如果缓慢减小 $a[i]$ 电荷值后的结果，通过以下循环结构代码实现图像的计算：

```
for (int t = 0; t < 100; t++)
{
    // 重新计算图中相应的值并重绘
    picture.show();
    a[1].increaseCharge(-2.0);
}
```



改变粒子的电荷值

423

3.2.25 用于复数的秒表。编写一个 `Stopwatch` 客户程序用于计算程序运行的时间开销，用以比较 `Mandelbrot` 程序不同实现方案的计算时间，一种方案是使用 `Complex` 实现的，另一种方案是直接操纵两个浮点数的方法实现的。具体来说，请创建一个只执行计算的 `Mandelbrot` 版本（删除引用 `Picture` 的代码），然后创建一个不使用 `Complex` 的程序版本，并计算它们运行时间的比值。

3.2.26 四元数。1843 年, 威廉·汉密尔顿爵士发现了一个复数的扩展, 称之为四元数。四元数是一个向量 $a = (a_0, a_1, a_2, a_3)$, 其计算公式如下:

- 模: $|a| = \sqrt{a_0^2 + a_1^2 + a_2^2 + a_3^2}$
- 共轭: a 的共轭是 $(a_0, -a_1, -a_2, -a_3)$
- 逆: $a^{-1} = (a_0/|a|^2, -a_1/|a|^2, -a_2/|a|^2, -a_3/|a|^2)$
- 和: $a+b = (a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3)$
- 点积: $a \times b = (a_0 b_0 - a_1 b_1 - a_2 b_2 - a_3 b_3, a_0 b_1 - a_1 b_0 - a_2 b_3 + a_3 b_2, a_0 b_2 - a_1 b_3 + a_2 b_0 + a_3 b_1, a_0 b_3 + a_1 b_2 - a_2 b_1 + a_3 b_0)$
- 商: $a/b = ab^{-1}$

为四元数创建一个数据类型 Quaternion, 并为你的代码创建一个测试客户程序。四元数将三维旋转的概念推广到四维, 它们被用于计算机图形学、控制理论、信号处理和轨道力学。

3.2.27 龙形曲线。写一个递归的 Turtle 客户程序 Dragon, 绘制龙形曲线 (见练习 1.2.35 和练习 1.5.9)。

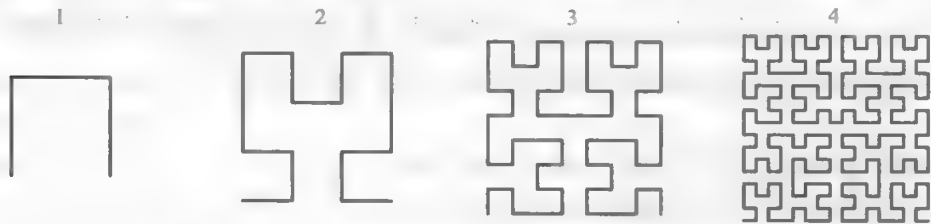
答案: 这些曲线最初是由 NASA 的三位物理学家发现的, 由马丁·加德纳 (Martin Gardner) 在 20 世纪 60 年代推广, 随后被克莱顿 (Michael Crichton) 在电影《侏罗纪公园》(Jurassic Park) 中使用。这个练习可以用非常紧凑的代码来实现, 基于一对直接从练习 1.2.35 中得到的相互作用的递归函数。其中一个函数 dragon() 用于绘制期望的龙形曲线, 另一个函数 nogard() 用于以相反的顺序绘制曲线。详情请参阅本书官网。

% java Dragon 15



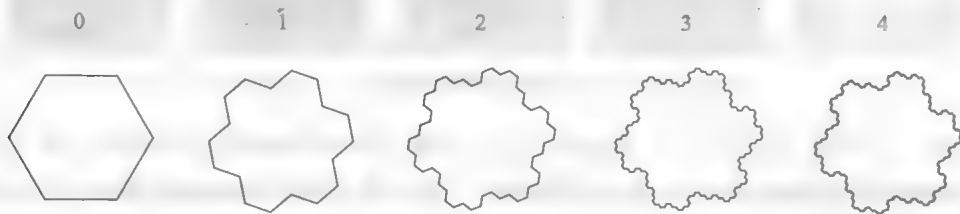
424

3.2.28 希尔伯特曲线。空间填充曲线是穿过每个点的单位正方形中的连续曲线。编写一个递归的 Turtle 客户程序创建这种递归图案, 这就是数学家大卫·希尔伯特 (David Hilbert) 在 19 世纪末定义的空间填充曲线。



部分答案: 设计一对相互递归的方法: hilbert() 用于遍历希尔伯特曲线, treblih() 则用于以相反的顺序遍历希尔伯特曲线。详情请参阅本书网站。

3.2.29 Gosper 岛。写一个递归的 Turtle 客户程序, 产生这些递归模式。



3.2.30 化学元素。为元素周期表中的条目创建一个数据类型 ChemicalElement。数据类型包含元素、原子序数、符号、原子量以及这些值的访问方法。然后创建一个数据类型 PeriodicTable, 它

从文件中读取数值来创建一个 ChemicalElement 对象矩阵（可以在本书网站上找到该文件及其格式描述），程序响应来自标准输入的查询，用户可键入一个分子式如 H₂O，程序响应并输出分子量。请分别为两种数据类型开发 API 并完成其实现。

425

- 3.2.31 数据分析。编写一个数据类型，用于运行实验，其控制变量的范围是 $[0, n)$ 内的一个整数，因变量是一个浮点数（例如，研究带一个整数参数的程序的运行时间会涉及此类实验）。实现以下 API：

```
public class Data
```

<code>Data(int n, int max)</code>	创建一个新的数据分析对象 对于 $[0, n)$ 中的 n 个整数值
<code>double addDataPoint(int i, double x)</code>	添加一个数据点 (i, x)
<code>void plotPoints()</code>	绘制所有的数据点

使用 StdStats 中的静态方法进行统计计算并绘制图形。编写一个测试客户程序，当网格大小增加时，绘制渗透原理实验的运行测试结果（渗透概率）。

- 3.2.32 股票价格。本书网站中的文件 DJIA.csv 包含了自有记录以来道琼斯工业平均指数的所有收盘价格，使用逗号分隔文件格式。请编写一个数据类型 DowJonesEntry，它可以保存表中的一个条目，包括日期、开盘价格、最高价格、最低价格、收盘价格等。然后再编写一个数据类型 DowJones，从文件中读取数据以创建 DowJonesEntry 对象数组，并提供计算任意时间跨度平均值的方法。最后，创建有趣的 DowJones 客户程序，产生数据的绘制图形。请读者发挥自己的创造力：这个过程会充满乐趣。
- 3.2.33 最大的赢家和最大的输家。编写一个 StockAccount 的客户程序，构建一个 StockAccount 对象的数组，计算每个账户的资金总额，并输出具有最大值和最小值的账户的报表信息。假设账户中的数据按照文中给出的格式保存在一个单独的文件中，这个文件包含了账户的信息。
- 3.2.34 牛顿迭代解方程之混沌情况。多项式 $f(z)=z^4-1$ 有四个根：1、-1、 i 和 $-i$ 。我们可以在复数平面上用牛顿法求解多项式的根： $z_{k+1}=z_k-f(z_k)/f'(z_k)$ 。其中， $f(z)=z^4-1$ ， $f'(z)=4z^3$ 。该方法根据初始点 z_0 收敛到四个根之一。请编写一个 Complex 和 Picture 的客户程序 NewtonChaos，它接受一个命令行参数 n ，并创建一个与以原点为中心的边长为 2 的正方形相对应的 $n \times n$ 图片。根据相应点收敛于四个根中的哪一个而为对应像素着色为白、红、绿或者蓝色（如果在 100 次迭代之后没有收敛则赋予像素点黑色）。
- 3.2.35 曼德布洛特的彩色绘图。创建一个包含 256 个整数三元组的文件，代表一些有趣的颜色值，然后使用这些颜色代替灰度值来绘制 Mandelbrot 中的每个像素。读取这些值，并创建包含 256 个 Color 值的数组，然后使用 mand() 的返回值作为数组索引下标从数组中提取这些值用于绘图。请通过试验在曼德布洛特集合中的不同位置尝试不同颜色选择，来制作出一张令人惊叹的图像。请参阅本书网站中提供的 mandel.txt。
- 3.2.36 朱莉娅集合。复数 c 的朱莉娅集合（Julia set）是与 Mandelbrot 函数相关的点集。这次我们固定 c 并改变 z ，而不是固定 z 而改变 c 。点 z 如果对于修改后的曼德布洛特函数能够有界，则属于朱莉娅集合。如果序列发散到无穷大，则不属于朱莉娅集合。所有相关的点位于以原点为中心的 4×4 的框中。当且仅当 c 在曼德布洛特集合中时， c 的朱莉娅集合才是连通的！编写一个程序 ColorJulia，它接收两个命令行参数 a 和 b ，使用上一个练习中描述的颜色表方法，绘制 $c=a+bi$ 的朱莉娅集合的彩色图像。

426

427

3.3 设计数据类型

创建数据类型的能力使得每个程序员都变成一个语言设计者。我们不再局限于处理内置于语言中的数据类型和相关操作，而是可以很容易地创建自己的数据类型并编写相应的客户程序。例如，Java 没有预定义的复数数据类型，但是你可以定义 `Complex` 数据类型，并编写客户程序，如 `Mandelbrot`。同样，Java 没有内置的海龟绘图工具，但是我们可以定义 `Turtle` 并编写客户程序以便立即利用这个抽象。即使 Java 确实包含了一个特定的工具，我们也可能更喜欢根据自己的特定需求创建单独的数据类型，就像我们自定义 `Picture`、`In`、`Out` 和 `Draw` 等类型一样。

基于上述观点，编写一个程序时，首要任务就是尽量理解我们需要的数据类型。程序开发在此时变为一项设计活动。在本节中，我们特别关注 API 的开发，它是任何程序开发的关键步骤。我们需要考虑不同的选择方案，了解其对客户程序和实现的影响，并不断优化设计方案，以在客户需求和可能的实现策略之间寻求平衡。

如果你选修了系统编程课程，那么会了解这种设计任务是构建大型系统的关键行为，并且在编写大型程序时，Java 及其他类似语言包含强大的高层机制，支持编写大型程序时的代码复用。这些机制中有许多是面向构建大型系统的专家而设计的，但其一般方法也适用于所有的程序员，其中一些机制适用于编写小型程序。

在本节中，我们将讨论封装、不变性和继承，关注这些设计理念在数据类型设计中的应用，以实现模块化程序设计，便于调试程序，提高编写清晰而正确的代码的效率。

在本节最后，我们将讨论一种重要的 Java 运行时机制，用于检查设计时的假设与实际运行时的条件是否匹配。这些功能对于开发可靠软件具有宝贵的价值。

设计 API 在 3.1 节，我们编写了使用 API 的客户程序。在 3.2 节，我们实现了 API。现在我们讨论设计 API 时的挑战。按照上述顺序来讨论这些主题是比较恰当的，因为程序设计花费的大部分时间主要在于编写客户程序。

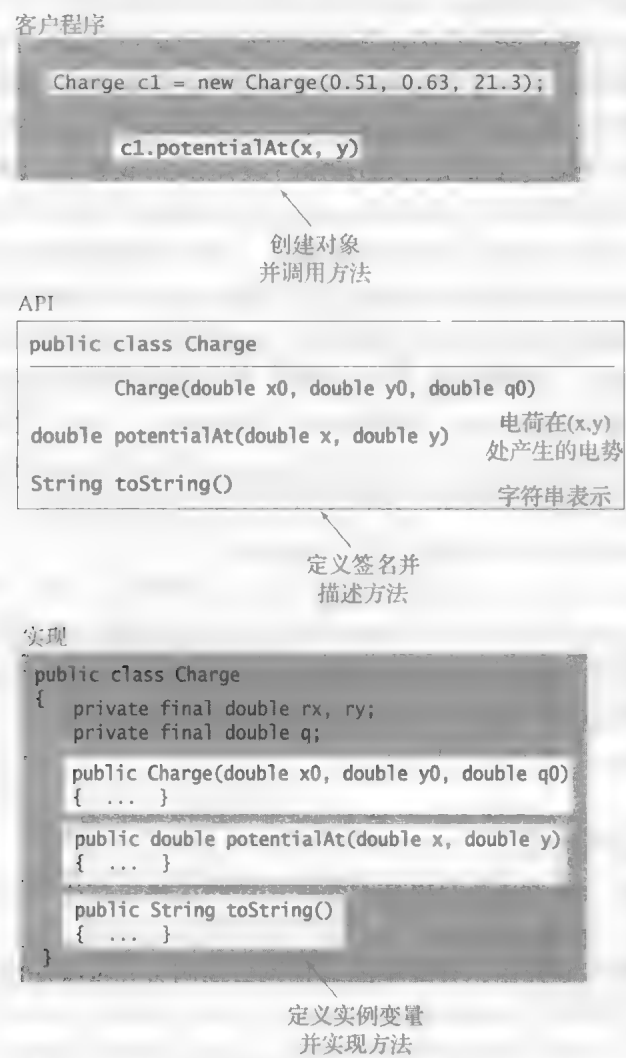
通常，构建软件最重要和最具挑战性的步骤是设计 API。这个任务需要不断实践、深思熟虑和多次迭代。但是，设计一个好的 API 所花的时间肯定会在调试或代码复用中得到及时的回报。

编写小型程序时斟酌 API 似乎没有必要，但应该考虑到你编写的程序也许将来在编写其他程序时可以复用——不是因为你知道你将使用哪段代码，而是因为你极可能希望将复用某些代码，但只是目前无法确定是哪些代码。

标准。通过与其他领域的类比，我们可以很容易地理解为什么编写 API 是如此重要。从火车铁轨到螺纹螺母和螺栓，到 MP3，再到无线电频率和 Internet 标准，我们了解到遵循一个常用的标准接口可以促进一项技术的广泛应用。Java 本身也是一个例子：用户编写的 Java 程序是 Java 虚拟机的客户程序，Java 虚拟机是一个在众多硬件和软件平台上实现的标准接口，因此 Java 程序可以“一处编译，多处执行”。通过使用 API 将客户程序与实现分离，在编写的每个程序中，我们都能得到标准接口带来的收益。

规范问题。数据类型 API 包含若干方法和这些方法所提供功能的简单描述。在理想情况下，API 将在规范（specification）中明确描述针对所有可能的参数会产生的行为及其副作用，然后编写软件来检查 API 的实现是否满足了所描述规范的要求。不幸的是，理论计算机科学的一个基本结论表明：这个目标事实上是无法实现的，这个问题被称为规范问题（specification problem）。简而言之，这种类型的规范说明应该使用形式语言描述，如程序设计语言（那么，按照这样的规则，规范也就变成了一段程序——译者注）。同时，确定两个程序是否执行了相

同计算的问题在数学上是不可解的（如果读者对这个观点感兴趣，可以通过学习一门称为“理论计算机科学”的课程，更多地了解不可解问题的本质及其对理解计算的本质起到的作用）。因此，我们只能通过示例进行非形式化的描述，就像我们写在 API 前后的描述文字一样。



面向对象的数据类型抽象

宽接口。宽接口是指包含的方法非常多的接口。设计 API 需要遵循的一个重要原则是避免宽接口。API 的规模一般会随着时间的推移自然而然地增长，因为向现有 API 添加方法很容易，然而删除方法又不破坏现有客户程序是很困难的。在某种情况下，宽接口也是合理的。例如，广泛使用的系统库（如 `String`）中就存在大量方法。多种技术有助于减少接口的有效宽度。一种途径是仅包含功能上相互正交的方法（即无法借助别的方法实现的方法——译者注）。例如，Java 的 `Math` 库包含正弦、余弦和正切的三角函数，但不包括正割和余割。

从客户代码开始。开发数据类型的主要目的之一是简化客户代码。因此，当开始设计一个 API 的时候，就关注客户代码是有意义的。通常，在编写实现之前编写客户代码是明智的。当我们发现客户代码过于烦琐时，解决这个问题的一种方法是写一个简化版本的代码来表示思考和计算的过程。或者，如果你已经编写了描述计算过程的简明注释，一个可能的出发点是考虑将注释转换成代码。

避免对表示方式的依赖。通常情况下，当开发一个 API 时，我们在心中会考虑其表示方式。毕竟，一个数据类型是一组值和定义在这组值上的一系列操作的集合，在不知道值的前提下讨论操作是没有意义的。然而，这与确定值的表示是不同的概念。我们使用数据类型的一个目的是，使得客户代码独立于数据的具体表现细节，从而实现代码简化。例如，我们的 `Picture` 和 `StdAudio` 的客户程序只关注图像和声音的简单抽象表示。这些抽象 API 的主要价值在于允许客户代码仅关注标准的抽象方法，而忽略这些抽象方法背后的大量细节。

API 设计中的陷阱。一个 API 可能难以实现，这意味着：实现 API 的过程十分困难或不可能完成；也可能太难使用，这是指创建的客户代码比没有使用 API 的更复杂。API 可能太窄，忽略了客户程序需要的方法；API 可能太宽，包括大量客户程序不需要的方法。API 可能太笼统，提供了没有用处的抽象；API 可能太具体，提供的抽象太详细或太分散以至于不可用。上述思想有时也可以总结成另一种设计理念：向客户程序提供其所需的方法，仅此而已。

当你第一次开始编程的时候，你只需要输入 `HelloWorld.java` 就可以了，除了其产生的结果之外，你不用理解它。从那时起，你通过模仿本书代码来学习程序设计，最终开发自己的代码来解决各种问题。设计 API 的过程也是一样的。本书、本书官网以及 Java 在线文档中包含许多可供学习和使用的 API，以增强读者自己设计和开发 API 的信心。

封装 通过隐藏信息将客户程序与接口实现分离的过程称为封装。实现的细节对于客户程序来说是隐藏的，实现也无从了解客户代码的详细信息，事实上客户代码是后来才编写的。

你可能已经猜到，我们一直在数据类型的实现中使用封装。在 3.1 节中，我们从“你不知道数据类型是如何实现的”这个设计理念开始。这个设计理念描述了封装的一个主要优点。我们认为这是非常重要的，因此没有向读者描述任何其他设计数据类型的方式。现在，我们将详细地解释使用封装的三个主要原因。使用封装的目的如下：

- 启用模块化编程。
- 便于调试。
- 使程序代码更清晰可读。

这些原因是联系在一起的（精心设计的模块化代码比在长程序中完全基于基本类型的代码更容易调试和理解）。

模块化编程。从第 2 章开始，开发中的编程风格强调了一种程序设计理念，即把大型程序分解为可以独立开发和调试的小型模块。这种方法通过将修改程序的影响限制在局部范围增强了软件的弹性，通过使用数据类型替代新实现提高了性能和准确度，改进了内存占用，更提高了代码复用。这一想法在许多环境中都有效。当使用系统模块的时候，我们经常从封装中获益。新版本的 Java 系统通常包含各种数据类型的新实现，但原有的 API 不会改变，因此原有的程序不需要修改就可以使用新的 Java 系统库。改进数据类型的实现具有强烈和持续的推动力，因为所有的客户程序都可以从改进中获益。模块化编程成功的关键在于保持模块之间的独立性。我们之所以这样做是认为 API 是客户程序和实现之间的唯一依赖关系。使用一个数据类型时无须理解其具体实现。这个设计理念的意思是数据类型的实现可以假设客户程序除了 API 之外对数据类型一无所知。

示例。例如，我们可以看程序 `Complex`（程序 3.3.1），它与程序 3.2.6 有相同的名称和 API，但对复数使用不同的表示形式。程序 3.2.6 使用笛卡儿表示，其中实例变量 `x` 和 `y` 表示复数 $x+iy$ 。程序 3.3.1 使用极坐标表示法，其中复数用实例变量 `r` 和 `theta`，以 $r(\cos\theta+i\sin\theta)$ 的形式表示。极坐标表示是有意义的，因为某些运算（如乘法和除法）中使用极坐标表示的复数更有效率。封装的思想是，我们可以用其他程序替换这些程序（无论出于何种原因）而无须更

改客户代码。两种实现之间的选择取决于客户程序。事实上，原则上对于客户程序的唯一区别应该是不同的执行性能。这种能力至关重要，可以给我们带来很多益处。其中最重要的一点是，它允许我们不断地改进软件：当我们开发一种更好的方法实现数据类型时，其所有的客户程序都将受益。每次安装新版本的软件系统（包括 Java 本身）时，都可以利用这项特性。

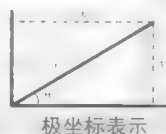
私有的。Java 语言对强制封装的实现使用私有访问修饰符 `private`。当我们声明一个实例变量（或方法）为 `private` 时，则不允许所有客户程序（另一个类中的代码）直接访问该修饰符修饰的实例变量（或方法）。因此，客户程序只能通过公共方法和构造函数（类的 API）来访问此数据类型。因此，可以修改 API 的实现，以使用不同的私有实例变量（或者重新组织私有实例方法），而且确信不会直接影响客户程序。Java 并不要求所有的实例变量都是私有的，但是我们在本书的程序中坚持使用这个约定。例如，如果 `Complex`（程序 3.2.6）中的实例变量 `re` 和 `im` 是公共的，则客户程序可以编写直接访问它们的代码。如果 `z` 引用一个 `Complex` 对象，则 `z.re` 和 `z.im` 可以引用这些值。但是，任何这样做的客户代码都完全依赖于该实现，违反了基本的封装规范。如果换成不同的实现，如程序 3.3.1 中的实现，会导致该代码无效。为了避免遇到这种情况，我们总是将实例变量设为私有。接下来，我们考察这个约定的一些影响。

433

程序3.3.1 复数（另一种实现方式）

```
public class Complex
{
    private final double r;
    private final double theta;
    public Complex(double re, double im)
    {
        r = Math.sqrt(re*re + im*im);
        theta = Math.atan2(im, re);
    }
    public Complex plus(Complex b)
    { // 返回这个数和b的和
        double real = re() + b.re();
        double imag = im() + b.im();
        return new Complex(real, imag);
    }
    public Complex times(Complex b)
    { // 返回这个数和b的乘积
        double radius = r * b.r;
        double angle = theta + b.theta;
        // 参考问答环节
    }
    public double abs()
    { return r; }
    public double re() { return r * Math.cos(theta); }
    public double im() { return r * Math.sin(theta); }
    public String toString()
    { return re() + " + " + im() + "i"; }
    public static void main(String[] args)
    {
        Complex z0 = new Complex(1.0, 1.0);
        Complex z = z0;
        z = z.times(z).plus(z0);
        z = z.times(z).plus(z0);
        StdOut.println(z);
    }
}
```

r 极径
theta 极角



这个数据类型实现了与程序3.2.6相同的API。它使用相同的实例方法但使用不同的实例变量。由于实例变量是私有的，所以这个程序可以用来代替程序3.2.6，而无须改变任何客户端代码。

```
% java Complex
-7.000000000000002 + 7.000000000000003i
```

434

规划未来。众多例子表明，许多严重的后果可以直接追溯到程序员没有封装其数据类型。

- **Y2K 问题。**在上一个千年，许多程序为了节省空间仅用两位数字来表示年份。这样的程序无法区分 1900 年和 2000 年。当 2000 年 1 月 1 日临近时，程序员开始争分夺秒地修正这种错误，以避免许多技术人员所预测的灾难性错误。
- **邮政编码。**1963 年，美国邮政局（USPS）开始使用 5 位邮政编码来改善邮件的分类和传送。程序员编写的软件假定邮政编码将永远保持在 5 位，并在程序中用一个 32 位的整数表示它们。1983 年，美国邮政推出了名为 ZIP+4 的扩展邮政编码，新邮政编码包含原始的 5 位邮政编码并附加额外的 4 位数字。
- **IPv4 与 IPv6。**互联网协议（IP）是电子设备通过互联网交换数据的标准。每个设备被分配一个唯一的整数或地址。IPv4 使用 32 位地址，支持约 43 亿个地址。由于互联网的爆炸性增长，新版本的 IPv6 使用 128 位地址，支持 2^{128} 个地址。

在这些情况下，如果程序没有正确地封装数据，则数据内部的改变将使得依赖于当前标准的大量客户代码（因为数据类型没有被封装）根本就不能按预期运行。上述案例的变动成本估计高达数亿美元！这就是没有封装数值所带来的巨大代价。这些困境对你来说可能很遥远，但是可以肯定的是，任何一个程序员（包括你自己）如果没有充分利用数据封装的预防措施，都有可能在标准改变的时候花费大量的时间和精力去修复失效代码。

我们的规则是：所有实例变量都是 `private` 访问修饰符定义的私有变量，以避免上述问题。如果你在实现数据类型（如年份、邮政编码、IP 地址或者其他任何数据类型）时采用了这种规范，就可以更改其内部表示而不影响客户程序的运行。数据类型实现对数据的具体表示了如指掌，同时对象存储数据。客户程序仅仅引用一个对象，无须知道细节。

435

限制潜在的错误。封装还有助于程序员确保其代码的运行结果符合预期。举一个例子，我们考虑另一个可怕的故事：在 2000 年的美国总统选举中，戈尔在佛罗里达州的 Volusia 县的电子投票机上收到了 -16 022 票。其原因是计数器变量没有正确地封装在投票机软件中！为了理解这个问题，我们将讨论根据下面的 API 实现的一个计数器（程序 3.3.2）：

```
public class Counter
```

<code>Counter(String id, int max)</code>	创建一个计数器，初始化为 0
<code>void increment()</code>	增加计数器的值，除非它的值已是最大值 max
<code>int value()</code>	返回计数器的值
<code>String toString()</code>	字符串表示

计数器数据类型的 API（参见程序 3.3.2）

这种抽象适用于许多情况，如电子投票机。Counter 封装了一个整数，并确保其对整数执行的唯一操作是将该整数递增 1。因此，其结果永远不会变成负数。数据抽象的目标是限制数据的操作。数据封装还能隔离数据的操作。例如，我们可以增加一个新的实现，使得计数器具有日志记录功能，`increment()` 可为每个投票记录一个时间戳或其他可用于检查一致性的信息。但是如果没有 `private` 修饰符，在投票机中可能会编写如下客户代码：

```
Counter c = new Counter("Volusia", VOTERS_IN_VOLUSIA_COUNTY);
c.count = -16022;
```

在使用 `private` 强制封装的程序设计语言中，类似这样的代码无法通过编译。如果没有

类似的保护，则戈尔的投票计数是负的。使用封装并不能完全解决投票安全问题，但这是一个好的开始。

436

程序 3.3.2 计数器

```
public class Counter
{
    private final String name;
    private final int maxCount;
    private int count;

    public Counter(String id, int max)
    { name = id; maxCount = max; }

    public void increment()
    { if (count < maxCount) count++; }

    public int value()
    { return count; }

    public String toString()
    { return name + ": " + count; }

    public static void main(String[] args)
    {
        int n = Integer.parseInt(args[0]);
        int trials = Integer.parseInt(args[1]);
        Counter[] hits = new Counter[n];
        for (int i = 0; i < n; i++)
            hits[i] = new Counter(i + "", trials);

        for (int t = 0; t < trials; t++)
            hits[StdRandom.uniform(n)].increment();
        for (int i = 0; i < n; i++)
            StdOut.println(hits[i]);
    }
}
```

name	计数器名称
maxCount	最大值
count	值

这个类封装了一个简单的整数计数器，为其分配一个字符串名称，并将其初始化为0（Java的默认初始化）。当客户端代码调用increment()时递增，当客户端调用value()时返回计数值，当调用toString()时返回包含其名称和值的字符串

```
% java Counter 6 600000
0: 100684
1: 99258
2: 100119
3: 100054
4: 99844
5: 100037
```

437

代码清晰度。精确地设计数据类型，可以使客户代码更清晰地表达计算过程，从而提高程序设计的质量。在 3.1 节和 3.2 节中，我们已经看到了很多客户代码的例子，在讨论 Histogram（程序 3.2.3）时就提到过这个问题。有精确数据类型的客户程序比没有的更加清晰，因为实例方法 addDataPoint() 的调用可以清楚地标识客户程序中的设计重点。可以观察到，良好的设计的一个关键之处在于，基于合理设计的抽象接口编写的代码几乎就可以当成文档。一些面向对象程序设计爱好者可能会争辩：如果 Histogram 使用 Counter 会更容易理解（参见练习 3.3.3），但这一点是存在争议的。

我们在本书中一直强调封装的优点。在设计数据类型的情景下，我们再次进行总结。封装可以实现模块化程序设计，允许我们：

- 独立开发客户程序和实现代码。
- 使用改进的实现版本而不会影响客户程序。
- 在尚未完成的程序上继续开发新程序（任何客户程序都可以基于 API 编写代码）。

封装还可以实现数据类型的操作隔离，这将：

- 在实现代码中添加一致性检查和其他调试工具。
- 使客户代码更加清晰。

438

一个正确实现的（被封装的）数据类型扩展了 Java 语言，使得任何客户程序都可以使用它。

不变性 正如 3.1 节末尾所定义的，如果一个数据类型的值一旦创建就不可更改，则该数据类型的对象是不可变的。不可变的数据类型是指该类型的所有对象均不可变。作为对比，可变数据类型指该类型对象的值被设计成可变的。在本章讨论的数据类型中，String、Charge、Color 和 Complex 都是不可变的，而 Turtle、Picture、Histogram、StockAccount 和 Counter 都是可变的。是否使数据类型不可变是一个基本的设计决策，取决于所开发的应用程序。

不可变类型。许多数据类型的目的是封装不会更改的数据，以便数据的行为与基本类型相同。例如，Complex 客户程序的程序员很可能期望编写代码 `z=z0`，这是一个合理的期望，就像使用 double 或者 int 变量那样使用两个 Complex 变量。但是如果 Complex 对象是可变的，并且 `z` 在执行赋值语句 `z=z0` 后改变了，那么 `z0` 的值也会改变（因为它们都是对同一个对象的引用）！这个出乎意料的结果被称为别名错误，对于许多面向对象程序设计的新手而言是一个意外。实现不可变类型的一个主要原因是我们可以在赋值语句中使用不可变对象（或者作为参数和函数返回值），而无需担心它们的值发生变化。

不可变的	可变的
String	Turtle
Charge	Picture
Color	Histogram
Complex	StockAccount
Vector	Counter
	Java 数组

可变类型。对于许多数据类型来说，抽象的目的是封装可更改的数据。Turtle（程序 3.2.4）就是一个很好的例子。我们使用 Turtle 是为了减轻客户程序跟踪变化值的压力。同样，对于 Picture、Histogram、StockAccount、Counter 和 Java 数组等类型，我们期望这些数据类型的值可以改变。当我们将一个 Turtle 作为一个参数传递给一个方法时，如 Koch，我们期望 Turtle 对象的值可以发生变化。

439

数组和字符串。在使用 Java 数组（可变）和 Java 字符串数据类型（不可变）时，作为客户程序的程序员，你已经了解了它们的区别。当你将一个 String 传递给一个方法时，你不必担心这个方法改变 String 中的字符序列，但是当你将一个数组传递给一个方法时，这个方法可以随意地更改数组元素的值。String 数据类型是不可变的，因为我们通常不希望字符串值可变；Java 数组是可变的，因为我们通常希望数组值是可变的。在其他一些情况下，我们可能想要可变的字符串（这是 Java 的 StringBuilder 数据类型的目的）以及不可变的数组（这是我们将在本节后面研究的 Vector 数据类型的目的）。

不可变的优点。一般来说，不可变数据类型更容易使用且一般不会误用，因为可改变对象值的代码范围远远小于可变类型。使用不可变数据类型的代码更容易调试，因为更容易在使用它们的客户代码中保证对象状态的一致性。当使用可变数据类型时，必须始终注意对象更改的时间和位置。

不可变的代价。不可变对象的不足之处在于必须为每一个值创建一个新的对象。例如，表达式 `z=z.times(z).plus(z0)` 会创建一个新的对象（`z.times(z)` 的返回值），然后使用该对象来调用 `plus()`，但无须保存该对象的引用。诸如 Mandelbrot（程序 3.2.7）这样的程序，创建了大量类似的中间对象。然而，这个开销通常是可控的，因为 Java 垃圾收集器通常优化这种情况。而且，如同 Mandelbrot 一样，当计算的关键是产生大量的中间值时，我们愿意承担此类开销。Mandelbrot 还创建了大量（不可变的）Color 对象。

final。我们可以使用 `final` 修饰符支持数据类型的强制不变性。当将一个实例变量声明为 `final` 的时候，变量只能被赋值一次，无论是在内联初始化语句中还是在构造函数中。任何其他可能尝试修改 `final` 变量值的代码都会导致编译错误。在我们的代码中，使用 `final` 修饰符的实例变量的值永远不会改变。这一策略作为不变值的保护，可以防止误更改，并使程序更容易调试。例如，我们不必在追踪中包含 `final` 变量，因为它的值永远不会改变。

引用类型。遗憾的是，只有当实例变量是基本类型而不是引用类型时，`final` 才能保证其不变性。如果引用类型的实例变量具有 `final` 修饰符，那么该实例变量（对象引用）的值永远不会改变——变量将始终引用同一个对象。但是，对象本身的值是可以改变的。例如，如果有一个数组实例变量声明为 `final`，我们是不能改变数组的（即改变其长度或类型），但可以改变数组中单个元素的值。因此，可能会出现别名错误。例如，以下代码并没有实现一个不可变的数据类型：

```
public class Vector
{
    private final double[] coords;
    public Vector(double[] a)
    {
        coords = a;
    }
    ...
}
```

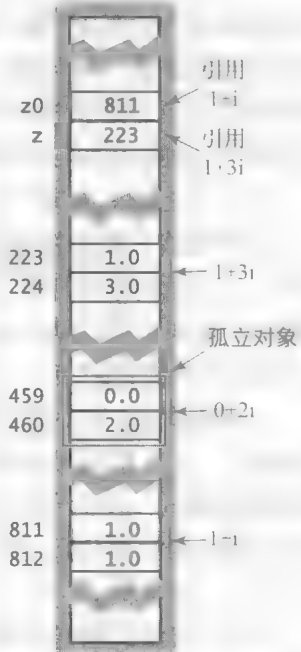
客户程序可以创建一个 `Vector` 对象，指定数组中的元素，并在创建 `Vector` 之后修改其元素（绕过 API）：

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 17.0; // coords[0]现在是17.0
```

实例变量 `coords[]` 标记为 `private` 和 `final` 类型，但 `Vector` 是可变的，因为实现代码中的引用指向与客户端相同的数组。如果客户代码改变了数组中的一个元素，那么这个改变也相应地出现在 `coords[]` 数组中，因为 `coords[]` 和 `a[]` 是别名。为了确保包含可变类型实例变量的数据类型的不可变性，我们需要创建一个本地拷贝，称为防御拷贝。接下来，我们研究这个拷贝的实现。

在任何数据类型的设计中都需要考虑不可变性问题。在理想情况下，应该在一个数据类型的 API 中指定其是否可变，以便客户程序知道对象值不可更改。实现一个不可变的数据类型可能是一种负担。对于复杂的数据类型，创建防御拷贝是一个挑战，确保没有任何实例

```
Complex z0;
z0 = new Complex(1.0, 1.0);
Complex z = z0;
z = z.times(z).plus(z0);
```

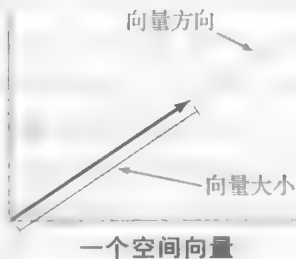


一个中间孤立对象

441 方法修改对象值是另一个挑战。

实例：空间向量 为了在实用的数学抽象情境中阐述上述设计理念，我们将讨论一种向量数据类型。与复数类似，向量抽象的基本定义众所周知，因为它已经在应用数学中发挥了100多年的重要作用。线性代数就是主要研究向量性质的数学领域。线性代数具有广泛应用的丰富而成功的理论，在社会科学和自然科学的各个领域都占据重要的地位。全面讨论线性代数显然超出本书的范围，但是很多重要的应用是基于基本和熟知的计算，所以向量和线性代数将在全书中多次提到（例如，1.6节中的随机冲浪者例子就是基于线性代数的）。因此，有必要将这种抽象封装为数据类型。

空间向量是一个具有大小和方向的抽象实体。空间向量提供了一个自然方法来描述物理世界属性，如力、速度、动量和加速度。指定一个向量的一种标准方法是在直角坐标系中从原点指向一个点的箭头：向量方向是从原点到点的射线方向，向量大小是箭头的长度（从原点到点的距离）。要指定一个向量，只需指定一个点即可。



上述概念可以扩展到任何维度的向量空间：一个 n 个实数的有序列表（一个 n 维空间点的坐标）能够指定一个 n 维空间中的向量。按照惯例，我们使用粗体来表示向量，在括号内用逗号分隔的数值或者带索引的变量名称（斜体，相同的字母，有索引下标）表示向量值。例如，我们可以用 x 来表示向量 $(x_1, x_2, \dots, x_{n-1})$ ，使用 y 来表示向量 $(y_1, y_2, \dots, y_{n-1})$ 。

API。向量的基本运算包括两个向量的加法、一个向量乘以一个标量、计算两个向量的点积、计算向量的大小和方向，其定义如下：

- 加法： $x+y=(x_0+y_0, x_1+y_1, \dots, x_{n-1}+y_{n-1})$
- 标量乘积： $\alpha x=(\alpha x_0, \alpha x_1, \dots, \alpha x_{n-1})$
- 点积： $x \cdot y=x_0 y_0+x_1 y_1+\dots+x_{n-1} y_{n-1}$
- 大小： $|x|=(x_0^2+x_1^2+\dots+x_{n-1}^2)^{1/2}$
- 方向： $x/|x|=(x_0/|x|, x_1/|x|, \dots, x_{n-1}/|x|)$

442

向量加法、标量乘积和方向的结果是向量，但大小和点积的结果是标量（实数）。例如，假设有 $x=(0, 3, 4, 0)$ ， $y=(0, -3, 1, -4)$ ，则 $x+y=(0, 0, 5, -4)$ ， $3x=(0, 9, 12, 0)$ ， $x \cdot y=-5$ ， $|x|=5$ ， $x/|x|=(0, 3/5, 4/5, 0)$ 。方向向量是一个单位向量：其大小为1。这些定义对应的API如下：

```
public class Vector
```

<code>Vector(double[] a)</code>	用给定的笛卡尔坐标创建一个向量
<code>Vector plus(Vector that)</code>	该向量和that的和
<code>Vector minus(Vector that)</code>	该向量和that的差
<code>Vector scale(double alpha)</code>	该向量与alpha的乘积
<code>double dot(Vector that)</code>	该向量和that的点积
<code>double magnitude()</code>	向量的模
<code>Vector direction()</code>	与该向量相同方向的单位向量
<code>double cartesian(int i)</code>	第i个笛卡尔坐标
<code>String toString()</code>	字符串表示

空间向量的API（参见程序3.3.3）

就像 Complex 的 API 一样，这个 API 没有明确指定数据类型为不可变对象，但是我们了解客户程序员（他们可能在数学抽象层面上思考）肯定希望对象不可变。

表示。依照惯例，我们开发实现的首选任务是选择一个数据的表示方式。在构造函数中使用数组保存直角坐标系各坐标值是一个明智的选择，但这不是唯一合理的选择。实际上，线性代数的基本原理之一是，满足特定条件的一组 n 个向量就可以用作坐标系统的基础（即坐标轴）：任何一个向量可以表示为一组 n 个向量的线性组合，这 n 个向量满足线性无关的特定条件。这种改变坐标系统的能力非常适合封装。大多数客户程序无须知道内部的具体表示，只需处理 Vector 对象和操作即可。如果可以保证，那么实现可以更换坐标系而不影响任何客户代码。

443

程序3.3.3 空间向量

```
public class Vector
{
    private final double[] coords;           coords[] | 笛卡儿坐标

    public Vector(double[] a)
    { // 做一个防御拷贝以确保对象不可变
      coords = new double[a.length];
      for (int i = 0; i < a.length; i++)
        coords[i] = a[i];
    }

    public Vector plus(Vector that)
    { // 该向量和that的和
      double[] result = new double[coords.length];
      for (int i = 0; i < coords.length; i++)
        result[i] = this.coords[i] + that.coords[i];
      return new Vector(result);
    }

    public Vector scale(double alpha)
    { // 向量与标量alpha的乘积
      double[] result = new double[coords.length];
      for (int i = 0; i < coords.length; i++)
        result[i] = alpha * coords[i];
      return new Vector(result);
    }

    public double dot(Vector that)
    { // 该向量和that的点积
      double sum = 0.0;
      for (int i = 0; i < coords.length; i++)
        sum += this.coords[i] * that.coords[i];
      return sum;
    }

    public double magnitude()
    { return Math.sqrt(this.dot(this)); }

    public Vector direction()
    { return this.scale(1/this.magnitude()); }

    public double cartesian(int i)
    { return coords[i]; }
}
```

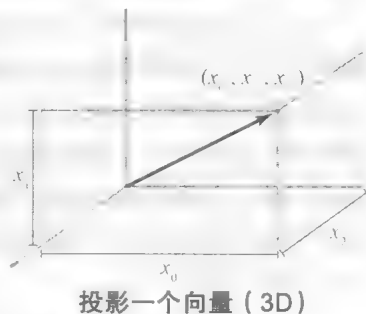
该实现将数学空间向量抽象封装在不可变的Java数据类型中。Sketch（程序3.3.4）和Body（程序3.4.1）是其典型的客户程序，实例方法minus()和toString()留作练习（练习3.3.4和练习3.3.14），测试客户程序是练习3.3.5。

444

实现。给定数据表示方法，实现这些操作的代码（程序3.3.3中的Vector）并不是一个困难的任务。构造函数创建了一个客户数组的防御拷贝，且没有任何方法可以给该拷贝赋值，因此Vector对象是不可变对象。cartesian()方法对于直角坐标表示十分容易：只需返回

数组中第 i 个坐标即可。它实际上实现了适用于任何 `Vector` 表示的数学函数：几何投影到第 i 个笛卡儿轴。

`this` 引用。在实例方法（或构造函数）中，`this` 关键字为我们提供了一种方法，用来引用调用实例方法（或构造函数）的对象。我们可以像使用任何其他对象引用一样（例如，调用一个方法、作为参数传递给方法或访问实例变量）来使用 `this`。例如，`Vector` 中的 `magnitude()` 方法使用 `this` 关键字的方式有两种：调用 `dot()` 方法和作为 `dot()` 方法的参数。因此，表达式 `vector.magnitude()` 等价于 `Math.sqrt(vector.dot(vector))`。一些 Java 程序员总是使用 `this` 来访问实例变量。这个策略使得代码很容易维护，因为它清楚地指明我们引用实例变量（而不是局部变量或参数变量）的时机。然而，这导致了 `this` 关键字的滥用，所以我们采取相反的策略，在代码中谨慎地使用 `this`。



当所有操作都能如此容易通过数组实现时，为什么要使用 `Vector` 数据类型呢？现在，这个问题的答案是显而易见的：启用模块化编程，不仅便于调试，而且可以使代码更清晰。`double` 型数组是一种低级的 Java 机制，允许对其元素进行各种操作。通过将我们的操作限制在 `Vector` 的 API 中（对于许多客户程序，这也是我们唯一需要的），可以简化设计、实现和维护客户程序的过程。由于 `Vector` 数据类型是不可变的，我们可以像使用基本数据类型一样使用它。例如，当将一个 `Vector` 对象传递给一个方法时，可以确信该对象的值不会改变（但在传递一个数组时并不能保证这一点）。使用 `Vector` 数据类型及其相关操作编写程序是一种简单而自然的方式，可以围绕此抽象概念开发大量数学知识相关的程序。

Java 语言支持定义对象之间的关系，这种关系的定义方式称为继承。软件开发人员广泛使用这些机制，因此，如果读者也学习软件工程课程，将会详细研究这些机制。通常，这些机制的实际使用超出了本书的知识范围，但我们将简要描述 Java 中的两种主要继承形式——接口继承和实现继承，在后面的几个例子中，我们可能会遇到它们。

接口继承（子类型化） Java 提供了 `interface`（接口）结构，用于指定若干个类必须实现的一组通用方法，这些类之间可能本来毫无关联。也就是说，`interface` 类描述的是一组方法的协议。我们把这种协议称为接口继承，因为实现类从接口类继承了部分 API。通过调用接口中的通用方法，我们能够编写可以操作不同类型的对象的客户程序。与大多数新的编程概念一样，刚开始有些混乱，但是在看过一些例子之后就会觉得容易理解了。

定义一个接口。作为一个启发性的例子，假设我们要编写代码来绘制任何实值函数。我们以前遇到过这样一个程序，即通过对函数在一个特定区间内用均匀间隔点进行采样来绘制一个特定函数。为了将这些程序扩展以处理任意函数，我们为单一变量的实值函数定义一个 Java 接口：

```
public interface Function
{
    public abstract double evaluate(double x);
}
```

接口声明的第一行类似于类声明，但是使用关键字 `interface` 而不是关键字 `class`。接口的主体包含一个抽象方法的列表。抽象方法是声明的方法，但不包含任何实现代码；它只包含方法签名，以分号结尾。`abstract` 修饰符将方法指定为抽象的。与 Java 类一样，我们必须

将 Java 接口保存在名称与接口名称匹配的文件中，并以 .java 作为扩展名。

446

实现一个接口。接口是一个类实现一组方法的协议。若要编写实现接口的类，必须执行两项操作。首先，必须在类声明中包含接口名称的 implements 子句。我们可以将其视为签署协议，承诺要实现接口中声明的每个抽象方法。其次，每个抽象方法都必须实现。例如，可以定义一个类，用于计算实数的平方，如下所示，这个类实现 Function 接口：

```
public class Square implements Function
{
    public double evaluate(double x)
    { return x*x; }
}
```

同样，可以定义一个计算高斯概率密度函数的类（见程序 2.1.2）：

```
public class GaussianPDF implements Function
{
    public double evaluate(double x)
    { return Math.exp(-x*x/2) / Math.sqrt(2 * Math.PI); }
}
```

如果我们未能实现接口中指定的任何一个抽象方法，将会出现编译时错误。相反，实现接口的类可能包括未在接口中指定的方法。

使用一个接口。接口是引用类型。我们可以像使用任何其他数据类型名称一样使用接口名称。例如，可以将变量的类型声明为接口的名称。这样做时，分配给该变量的任何对象都必须是实现这个接口的类的实例。例如，Function 类型的变量可以存储 Square 或 GaussianPDF 类型的对象，但不能存储 Complex 类型的对象。

```
Function f1 = new Square();
Function f2 = new GaussianPDF();
Function f3 = new Complex(1.0, 2.0); // 编译时错误
```

即使实现类定义了其他方法，接口类型的变量也只能调用在接口中声明的方法。

447

当接口类型的变量调用接口中声明的方法时，Java 知道程序在调用哪个方法，因为调用对象的类型是明确的。例如，f1.evaluate() 将调用 Square 类中定义的 evaluate() 方法，而 f2.evaluate() 将调用 GaussianPDF 类中定义的 evaluate() 方法。这个强大的编程机制被称为多态 (polymorphism) 或动态分派 (dynamic dispatch)。

为了体会使用接口和多态的好处，我们回到在区间 $[a, b]$ 中绘制函数 f 的图像的应用。如果函数 f 足够平滑，我们可以在区间 $[a, b]$ 的 $n+1$ 个均匀间隔的点上对函数进行采样，并使用 StdStats.plotPoints() 或 StdStats.plotLines() 显示结果。

```
public static void plot(Function f, double a, double b, int n)
{
    double[] y = new double[n+1];
    double delta = (b - a) / n;
    for (int i = 0; i <= n; i++)
        y[i] = f.evaluate(a + delta*i);
    StdStats.plotPoints(y);
    StdStats.plotLines(y);
}
```

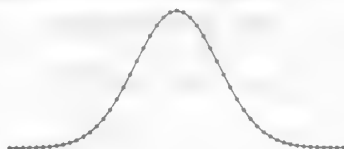
使用接口类型 Function 声明变量 f 的好处是，对于实现 Function 接口的任何数据类型的对象 f ，包括 Square 或 GaussianPDF，都使用相同的方法调用 f.evaluate()。因此，我们不

需要为每种类型编写重载方法——可以为许多类型复用相同的 `plot()` 函数！这种不修改客户代码而绘制任何函数的能力，是接口继承的一个有说服力的例子。

```
Function f1 = new Square();
plot(f1, -0.6, 0.6, 50);
```



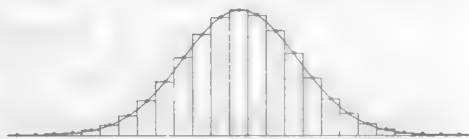
```
Function f2 = new GaussianPDF();
plot(f2, -4.0, 4.0, 50);
```



绘制函数图

用函数计算。有时，特别是在科学计算中，经常需要对函数进行计算：我们需要处理微分函数、整数函数、求函数的根等。在某些编程语言（被称为函数式编程语言）中，这个需求与语言的底层设计是一致的，其使用函数计算来大大简化客户代码。遗憾的是，函数不是 Java 中最重要的对象。但是，正如刚才的例子 `plot()` 一样，我们可以使用 Java 接口来实现一些相同的目标。

例如，研究在区间 (a, b) 中的正实数函数 f （曲线下区域）的黎曼积分的估计问题。这个计算被称为正交（quadrature）或数值积分（numerical integration）。已经有一些方法可用以求积分。在这些方法中，最简单的也许就是矩形规则，我们通过计算曲线下 n 个等宽矩形的总面积来近似求得积分值。下面定义的 `integrate()` 函数使用矩形规则将区间划分为 n 个矩形，用以计算区间 (a, b) 中实值函数 f 的积分：



近似积分

```
public static double integrate(Function f,
                               double a, double b, int n)
{
    double delta = (b - a) / n;
    double sum = 0.0;
    for (int i = 0; i < n; i++)
        sum += delta * f.evaluate(a + delta * (i + 0.5));
    return sum;
}
```

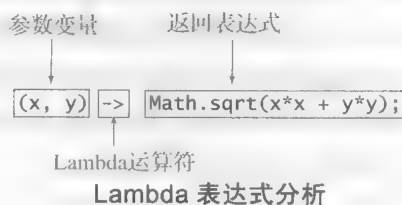
x^2 的不定积分是 $x^3/3$ ，所以区间 $0 \sim 10$ 的定积分是 $1000/3$ 。调用 `integrate(new Square(), 0, 10, 1000)` 将返回 `333.33324999999996`，如果我们只处理结果的六位有效数字，这就是正确答案。类似地，调用 `integrate(new GaussianPDF(), -1, 1, 1000)` 返回 `0.6826895727940137`，这是只处理七位有效数字时的正确答案（如果需要，可以回顾高斯概率密度函数和程序 2.1.2）。

正交法并不总是计算函数值的最有效或最精确的方法。例如，程序 2.1.2 中的 `Gaussian.cdf()` 函数是实现高斯概率密度函数的一种更快、更精确的方法。然而，这个方法的优点是对任何函数都可以使用，只是需要一定的技术条件以保证曲线足够光滑。

Lambda 表达式。我们刚刚研究的用于计算函数的语法有些笨拙。例如，为了实现绘制或积分，为每个函数定义一个新类，用于实现 `Function` 接口，这是一件很烦琐的事。在这种情况下，为了简化语法，Java 提供了一个强大的函数编程功能，称为 Lambda 表达式。我们应该将 Lambda 表达式看作一段可以在以后传递和执行的代码。在最简单的形式中，Lambda 表达式由三个元素组成：

- 参数变量列表（用逗号分隔，并用圆括号括起来）。
- Lambda 运算符 `->`。
- 一个单独的表达式，它是 Lambda 表达式返回的值。

例如，Lambda 表达式 `(x, y) -> Math.sqrt(x * x + y * y)` 实现了求直角三角形斜边的函数。当只有一个参数时，括号是可选的，所以 Lambda 表达式 `x -> x * x` 实现平方函数，`x -> Gaussian.pdf(x)` 实现高斯概率密度函数。



Lambda 表达式的主要功能是以简洁的方式来实现函数接口（具有单个抽象方法的接口）。

具体而言，可以在需要函数接口对象的任何地方使用 Lambda 表达式。例如，可以通过调用 `integrate(x -> x * x, 0, 10, 1000)`，从而绕过定义 `Square` 类的需求。我们不需要明确声明 Lambda 表达式实现了 `Function` 接口，只需要单个抽象方法的签名与 Lambda 表达式（相同数量的参数和类型）兼容，Java 就会在上下文中推断出这个表达式。在这种情况下，Lambda 表达式 `x -> x * x` 与抽象方法 `evaluate()` 兼容。

表达式
<code>new Square()</code>
<code>new GaussianPDF()</code>
<code>x -> x * x</code>
<code>x -> Gaussian.pdf(x)</code>
<code>x -> Math.cos(x)</code>

实现 `Function` 接口的典型表达式

内置接口。我们将在本书后面研究 Java 内置的三个接口。在 4.2 节中，我们将研究 Java 的 `java.util.Comparable` 接口，其包含一个抽象方法 `compareTo()`。`compareTo()` 方法通过比较相同类型的对象来定义一个自然顺序，如字符串的字母顺序以及整数和实数的升序，这使得我们能够编写代码对对象数组进行排序。在 4.3 节中，我们将使用接口来使客户程序能够循环访问集合中的项，而不依赖于底层表示形式。Java 为此提供了两个接口：`java.util.Iterator` 和 `java.lang.Iterable`。

基于事件的编程。接口继承值的另一个强大示例是它在基于事件的编程中的使用。例如，考虑扩展 `Draw` 来响应用户输入的问题，如响应用户点击鼠标或敲击键盘。一种方法是定义一个接口来指定在用户输入发生时，`Draw` 应该调用哪个或哪些方法。描述性术语回调用来描述一种基于接口的调用，这种调用从一个类中的方法到另一个类中的方法。你可以在本书官网上找到一个示例接口 `DrawListener`，以及如何编写代码以响应用户在 `Draw` 中的鼠标点击和键盘敲击。我们可以发现实现以下目标很容易：创建一个 `Draw` 对象，并为其添加一个回调方法，当用户输入事件（点击鼠标或敲击键盘）发生时，将调用这一回调方法，并将用户键入的字符或鼠标单击的位置传递给方法。编写交互式代码很有趣，但很有挑战性，因为你必须考虑到所有可能的用户输入操作。

接口继承是一种高级编程概念，被许多经验丰富的程序员所接受，因为它能够在不牺牲封装的情况下复用代码。接口继承支持的函数式编程风格在某些方面具有争议，但是 Lambda 表达式和类似的结构可以追溯到早期编程，并且许多现代编程语言也有这种方式。这种风格的支持者坚信我们应该专门使用和讲授函数式编程。本书一开始没有重视这种编程风格，因为我们所遇到的大量代码并不是这种风格的。我们在这里介绍这种方式，是因为每个程序员都需要了解这种方式，并在合适的时候使用。

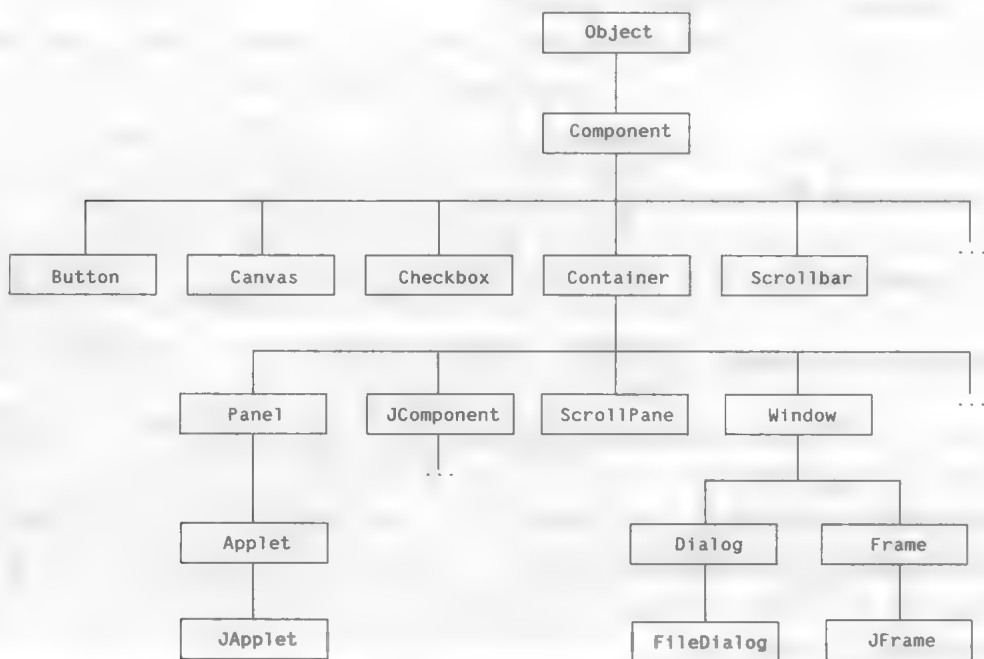
实现继承（子类化）Java 还支持另一种被称为子类化的继承机制。其目的是定义一个新的类（称为子类或派生类），从另一个类（称为父类或基类）继承其实例变量（状态）和实例方法（行为），从而实现代码复用。通常，子类重新定义或重写父类中的某些方法。我们

450

451

把这种机制称为实现继承，因为这时一个类继承了另一个类的代码。

系统程序员使用子类化来构建所谓的可扩展库——一个程序员（包括读者自己）可以向另一个程序员（或者可能是一个系统程序员团队）构建的库中添加方法，从而在一个潜在的巨大的库中有效地复用代码。这种方法被广泛使用，特别是用于用户界面的开发，以便复用那些为用户提供实现所期望的各种功能（窗口、按钮、滚动条、下拉菜单、剪切粘贴、访问文件等）的代码。



GUI 组件的子类继承层次结构（部分）

在系统程序员之间，继承的使用是存在争议的，因对其是否优于子类型（subtyping，即对 interface 的继承）是有争议的。在本书中，我们没有使用继承，因为在两个方面继承是与封装对立的。首先，任何父类的改变都会影响所有的子类。子类不能独立于父类开发，实际上，子类完全依赖于父类。这个问题被称为脆弱的基类（fragile base class）问题。其次，子类代码可以访问父类中的实例变量，从而违背父类代码的目标。例如，尽管像 Vector 这样的类的设计者可能非常小心地确保 Vector 不可变，但是一个能够完全访问这些实例变量的子类，却可以毫不费力地修改它们。

Java 的 Object 父类。某些子类的痕迹已经被内置到 Java 中，因此继承是无法避免的。具体而言，每个类都是 Java 的 Object 类的一个子类。这个结构能够实现这样的“约定”：每个类都包含 toString()、equals()、hashCode() 和其他几个方法的实现。每个类都从 Object 继承这些方法。在使用 Java 进行编程时，经常需要重写这些方法中的一个或多个。

```
public class Object
```

```
String toString()      这个对象的字符串表示形式
```

```
boolean equals(Object x) ... 这个对象是否等于x?
```

```
int hashCode()        这个对象的散列码
```

```
Class getClass()      这个对象的类
```

被所有类继承的方法（本书中用到的）

字符串转换。每个 Java 类都会继承 toString() 方法，因此任何一个客户程序都可以调用任何一个对象的 toString() 方法。与 Java 接口一样，Java 知道要调用哪个 toString() 方法（多态），因为 Java 知道调用对象的类型。当另一个操作数是字符串时，这个约定是 Java 自动将字符串连接运算符 “+” 的一个操作数转换为字符串类型的基础。例如，如果 x 是任意一个对象的引用，则 Java 自动将表达式 "x="+x 转换为 "x="+x.toString()。如果一个类没有重写 toString() 方法，那么 Java 会调用继承的 toString() 来实现，但这通常是没有用的（通常是对象内存地址的字符串表示形式）。因此，在我们开发的每个类中重写 toString() 方法是一个很好的编程习惯。

453

等式。两个对象相等意味着什么？如果使用 x 和 y 作为对象来测试等式 (x==y)，我们测试的是它们是否具有相同的标识：对象引用是否相同。例如，考虑右图中的代码，该代码创建了由三个变量 c1、c2 和 c3 引用的两个 Complex 对象（程序 3.2.6）。如图所示，c1 和 c3 引用了同一个对象，c2 引用的对象与它们不同。因此，(c1==c3) 为 true，但 (c1==c2) 为 false。这被称为引用相等，但一般情况下这不是客户想要的功能。

一般客户端希望测试数据类型的值（对象状态）是否相同。这被称为对象相等。Java 包含 equals() 方法（被所有类继承）。例如，String 数据类型以自然方式重写此方法：如果 x 和 y 引用 String 对象，则当且仅当两个字符串对应于相同的字符序列，x.equals(y) 才为 true（而不是取决于它们是否引用相同的 String 对象）。

Java 的约定是，对于所有的对象引用 x、y 和 z，equals() 方法必须通过满足以下三个自然属性实现相等关系：

- 自反性：x.equals(x) 是 true。
 - 对称性：当且仅当 y.equals(x) 为 true，x.equals(y) 才为 true。
 - 传递性：如果 x.equals(y) 为 true 且 y.equals(z) 为 true，则 x.equals(z) 为 true。
- 另外，以下两个属性必须保持：
- 对 x.equals(y) 的多次调用返回相同的真值，前提是两个调用之间不能修改对象。
 - x.equals(null) 返回 false。

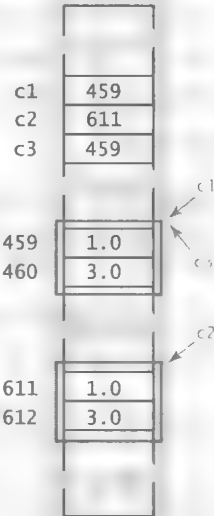
通常，当自定义数据类型时，需要重写 equals() 方法，因为继承的实现是引用相等。例如，当且仅当两个复数对象的实部和虚部相等，我们才认为这两个复数相等。以下实现可以完成这项工作：

454

```
public boolean equals(Object x)
{
    if (x == null) return false;
    if (this.getClass() != x.getClass()) return false;
    Complex that = (Complex) x;
    return (this.re == that.re) && (this.im == that.im);
}
```

这个代码比我们想象的复杂，因为 equals() 的参数可以是任何类型（或 null）对象的引用，因此我们总结每个语句的作用：

```
Complex c1, c2, c3;
c1 = new Complex(1.0, 3.0);
c2 = new Complex(1.0, 3.0);
c3 = c1;
```



两个对象的三个引用

- 如果参数为 `null`，则第一个语句返回 `false`。
- 如果两个对象的类型不同，则第二条语句使用继承的方法 `getClass()` 返回 `false`。
- 第二条语句使得第三条语句中的类型转换一定能够成功。
- 最后一条语句通过比较两个对象的对应实例变量来实现相等测试的逻辑。

我们可以使用这个实现作为模板——一旦实现了一个 `equals()` 方法，读者会发现实现另一个很简单。

散列法。我们现在讨论一个与相等测试相关的基本操作，称为散列（hashing），用于将一个对象映射为一个整数（称为散列码）。这种运算非常重要，由一个名为 `hashCode()` 的方法来处理，这个方法被所有的类继承。Java 的约定是对于所有的对象引用 `x` 和 `y`，`hashCode()` 方法必须满足以下两个属性：

- 如果 `x.equals(y)` 为 `true`，则 `x.hashCode()` 等于 `y.hashCode()`。
- 多次调用 `x.hashCode()` 返回相同的整数，前提是对象在调用之间不被修改。

例如，在下面的代码片段中，`x` 和 `y` 引用的是相同的 `String` 对象——`x.equals(y)` 是 `true`，所以它们的散列码必须相同；`x` 和 `z` 引用的是不同的 `String` 对象，所以我们期望它们的散列码不同。

```
String x = new String("Java"); // x.hashCode() 是 2301506
String y = new String("Java"); // y.hashCode() 是 2301506
String z = new String("Python"); // z.hashCode() 是 -1889329924
```

在一个典型的应用程序中，我们使用散列码把一个对象 `x` 映射为一个小范围内的整数，

[455] 比如在 0 到 $m-1$ 之间，使用如下 `hash` 函数：

```
private int hash(Object x)
{ return Math.abs(x.hashCode() % m); }
```

调用 `Math.abs()` 是为了确保返回值不是一个负整数（在 `x.hashCode()` 为负的情况下）。我们可以将散列函数值作为一个长度为 m 的数组的整数索引（在程序 3.3.4 和程序 4.4.3 中，这个操作的效果是明显的）。按照惯例，值相等的对象必须具有相同的散列码，因此它们也具有相同的 `hash` 函数值。值不相等的对象可以拥有相同的 `hash` 函数值，但我们期望 `hash` 函数将 n 个典型对象分成数量大致相等的 m 个组。许多 Java 的不可变数据类型（包括 `String`）都包含 `hashCode()` 的实现，都可以按照合理的方式分布对象。

为数据类型编写一个好的 `hashCode()` 实现要求结合科学与工程技巧，这超出了本书的范围。作为替代方法，我们描述了在 Java 中实现散列函数的一个简单有效的方法，并且适用于各种情况：

- 确保数据类型是不可变的。
- 导入 `java.util.Objects` 类。
- 通过比较所有重要的实例变量来实现 `equals()`。
- 通过将所有重要的实例变量作为静态方法

`Objects.hash()` 的参数来实现 `hashCode()`。

静态方法 `Objects.hash()` 为其参数序列生成一

```
import java.util.Objects;

public class Complex
{
    private final double re, im;
    ...

    public boolean equals(Object x)
    {
        if (x == null) return false;
        if (this.getClass() != x.getClass())
            return false;
        Complex that = (Complex) x;
        return (this.re == that.re)
            && (this.im == that.im);
    }

    public int hashCode()
    { return Objects.hash(re, im); }

    public String toString()
    { return re + " + " + im + "i"; }
}
```

重写 `equals()`、`hashCode()` 和 `toString()` 方法

个散列码。例如，如下是 Complex 数据类型的 hashCode() 实现（程序 3.2.1）和我们刚刚讨论的 equals() 实现：

```
public int hashCode()  
{ return Objects.hash(re, im); }
```

456

包装类型。继承的主要好处之一是代码复用。但是，这种代码复用仅限于引用类型（而不是基本类型）。例如，表达式 x.hashCode() 对于任何对象引用 x 都是合法的，但如果 x 是基本类型的变量，则编译时会产生错误。对于希望将基本类型的值表示为对象的情况，Java 提供了内置的引用类型（称为包装类型），8 种基本类型都有其对应的包装类型，如包装类型 Integer 和 Double 分别对应于 int 和 double。包装类型的对象将基本类型的值“包装”到对象中，以便我们可以使用实例方法，如 equals() 和 hashCode()。每种包装类型都是不可变的，并且包括实例方法（如用于在数值上比较两个对象的 compareTo()）和静态方法（如用于将字符串转换为基本类型的 Integer.parseInt() 和 Double.parseDouble()）。

基本数据类型	包装类型
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

自动包装和取消包装。Java 会自动在包装类型的对象和相应的基本数据类型值（基于赋值语句、方法参数和算术 / 逻辑表达式）之间进行转换，以便我们可以编写如下所示的代码：

```
Integer x = 17; // 自动包装(int ->Integer)  
int a = x;      // 取消包装(Integer ->int)
```

在第一条语句中，Java 自动将 int 值 17 强制转换（自动包装）为 Integer 类型的对象，然后将其分配给变量 x。同样，在第二条语句中，Java 在将该值赋给变量 a 之前，自动将 Integer 对象转换（取消包装）为 int 类型的值。在编写代码时，自动包装和取消包装是很方便的功能，但这些功能涉及大量的处理，可能会影响性能。

为了代码清晰和性能，我们尽可能使用基本类型数值进行计算。但是在第 4 章中，我们将遇到几个有趣的例子（特别是用于存储对象集合的数据类型），利用包装类型和自动 / 取消包装，我们能够无须修改地把为引用类型开发的代码复用于基本类型的处理。

457

应用：数据挖掘 为了在应用程序的上下文阐明这一节中讨论的一些概念，接下来我们将介绍一种软件技术——数据挖掘，数据挖掘描述了通过搜索大量信息来发现模式的过程，该技术被证明对于解决一些艰巨的挑战任务非常重要。数据挖掘技术可以作为提高 Web 搜索结果的质量、多媒体信息检索、生物医学数据库、基因组研究、改进许多领域的学术研究、商业应用创新、研究犯罪分子的计划，以及许多其他用途的基础。因此，研究人员对数据挖掘有着浓厚的兴趣和广泛的研究。

我们可以直接访问计算机中的数千个文件，并可以间接访问 Web 上的数十亿文件。正如你所知道的，这些文件非常多样化，有商业网页、音乐和视频、电子邮件、程序代码以及各种其他信息。为了简单起见，我们只关注文本文件（尽管我们研究的方法同样适用于图像、音频和各种其他文件）。虽然限制文件类型为文本，但文本文件的文档类型也有很大的差异。作为参考，读者可以在本书官网上找到这些文档：

文件名称	描述	示例文本
Constitution.txt	法律文件	... of both Houses shall be determined by ...
TomSawyer.txt	美国小说	..."Say, Tom, let ME whitewash a little." ...
HuckFinn.txt	美国小说	...was feeling pretty good after breakfast...
Prejudice.txt	英国小说	... dared not even mention that gentleman...
Picture.java	Java代码	...String suffix = filename.substring(file...
DJIA.csv	财政数据	...01-Oct-28,239.43,242.46,3500000,240.01 ...
Amazon.html	网页源码	...<table width="100%" border="0" cellspac...
ACTG.txt	病毒基因组	...GTATGGAGCAGCAGACGCGCTACTTCGAGCGGAGGCATA...

一些文本文档

458

我们的研究目标是寻找通过文件内容搜索文件的有效方法。解决这个问题一个有效方法是将每个文档与一个称为摘要 (sketch) 的向量相关联, 这个摘要要是其内容的浓缩表示。其基本思想是文档摘要应该捕获文档足够的统计信息, 即不同的文档具有不同的文档摘要, 相似的文档具有相似的文档摘要。这种方法能够区分一本小说、一个 Java 程序和一个基因组, 这也许并不奇怪, 但你可能会惊讶地发现文档摘要居然可以分辨出不同作者写的小说之间的区别, 甚至可以作为许多其他细微搜索的基础。

首先, 我们需要文本文档的抽象。什么是文本文档? 文本文档可以支持哪些操作? 这些问题的答案涉及该如何设计并编写代码。为了达到数据挖掘的目的, 第一个问题的答案很明显, 文本文档是由一个字符串定义的。第二个问题的答案是, 我们需要能够计算出一个数字, 以衡量一个文档和任何其他文档之间的相似度。基于这些考虑, 得到以下 API:

```
public class Sketch
{
    Sketch(String text, int k, int d)
    double similarTo(Sketch other)
    String toString()
}
```

Sketch 的 API (见程序 3.3.4)

构造函数的参数为一个字符串以及两个控制文档摘要质量和大小的整数。客户程序可以使用 similarTo() 方法来确定两个文档摘要的相似度, 相似度的范围在 0 (不相似) 到 1 (相似) 之间。toString() 方法主要用于调试。这种数据类型提供了实现相似性度量和实现通过使用度量来查找文档的客户程序之间的良好分离。

459

计算文档摘要。计算文档摘要是第一个挑战。我们将使用一系列实数 (或一个 Vector) 来表示文档摘要。但是, 文档摘要应该包含哪些信息? 如何计算这些信息? 目前为止已经有许多不同的方法, 研究者仍然在积极寻求解决这个高效算法。我们的 Sketch 实现 (程序 3.3.4) 使用简单的频率统计方法。构造函数包含两个参数: 整数 k 和向量维度 d 。算法扫描文档并检测文档中所有的 k -gram——从每个位置开始的长度为 k 的子字符串。文档摘要最简单的形式是表示字符串中 k -gram 相对频率的向量: 对于每一种可能的 k -gram, 给出文档中出现 k -gram 的数量。例如, 假设我们在基因组数据中使用 $k=2, d=16$ (存在 4 种可能的字符值, 因此有 $4^2=16$ 种可能的 2-gram)。

2-gram AT 在字符串 ATAGATGCATAGCGCATAGC 中出现 4 次, 因此在例子中对应 AT 的向量元素将是 4。为了构造频率向量, 我们需要把 16 个可能的 k -gram 中的每一个转换为一个 0 到 15 之间的整数 (该函数被称为散列函数)。对于基因组数据, 请参考一个简单的练习 (参见练习 3.3.28)。然后, 我们可以通过扫描文档计算一个数组来构建频率向量, 每个 k -gram 对应数组的一个元素。看起来我们忽略了 k -gram 的顺序而丢失了信息, 但明显的事实是信息内容中顺序的重要性低于其频率。如果我们要考虑顺序的问题, 可以使用马尔可夫模型, 与

比较文档摘要。第二个挑战是计算两个文档摘要之间的相似度。存在很多不同的方法来比较两个向量，最简单的可能就是计算它们之间的欧氏距离。给定向量 x 和 y ，欧氏距离由下面的式子定义

$$|x-y|=((x_0-y_0)^2+(x_1-y_1)^2+\cdots+(x_{d-1}-y_{d-1})^2)^{1/2}$$

当 $d=2$ 或 $d=3$ 时，读者很熟悉这个公式。对于 Vector，欧几里得距离（欧氏距离）的计算十分简单。如果 x 和 y 是两个 Vector 对象，则 $x.minus(y).magnitude()$ 是它们之间的欧氏距离。如果文档相似，我们期望其文档摘要之间的欧氏距离很小。另一个被广泛使用的相似性度量（称为余弦相似性度量）更简单：由于文档摘要要是具有非负坐标的单位向量，因此其点积

$$x \cdot y = x_0y_0 + x_1y_1 + \cdots + x_{d-1}y_{d-1}$$

是 0 到 1 之间的数值。在几何意义上，这个值是两个向量夹角的余弦（见练习 3.3.10）。文档越相似，我们期望这个度量越接近 1。

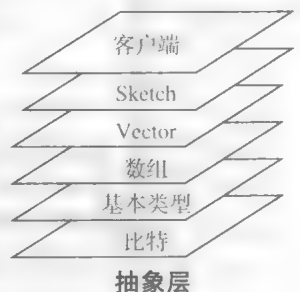
比较所有的文档对。CompareDocuments（程序 3.3.5）是一个简单但有效的 Sketch 客户程序，提供用于解决以下问题的信息：给定一组文档，找到最相似的两个文档。因为这份规范的描述具有一定的主观性，我们把它实现为 CompareDocuments，它的作用是计算输入列表中所有文档两两之间的余弦相似性度量值，并把结果打印出来。对于中等大小的 k 和 d ，文档摘要在刻画样本文件的特征方面做得很好。结果表明基因组数据、财务数据、Java 代码和 Web 源代码与法律文件、小说之间有着显著差异，小说《汤姆·索亚》和《哈克贝利·芬恩》之间的相似度比《傲慢与偏见》的相似度更高。比较文学的研究人员可以使用这个程序来发现文本之间的关系。教师也可以使用这个程序来检测一组学生提交的作业是否存在抄袭行为（事实上，许多教师经常使用这些程序）。生物学家可以使用这个程序来发现基因组之间的关系。你可以在本书官网上找到许多文档（或者也可以试试你自己收藏的文件），以测试各种参数设置下

```
% more documents.txt
Constitution.txt
TomSawyer.txt
HuckFinn.txt
Prejudice.txt
Picture.java
DJIA.csv
Amazon.html
ACTG.txt
```

462 CompareDocuments 程序的有效性。

搜索相似的文档。Sketch 的另一个客户程序是使用文档摘要在大量文档中搜索以查找与给定文档类似的文档。例如，Web 搜索引擎使用类似的客户程序向用户展示与之前访问的网页类似的网页，在线图书商家使用此类客户程序推荐与用户购买过的图书类似的图书，社交网站使用此类客户程序来识别与用户个人兴趣类似的人。由于 In 构造函数可以接收网址而不需要文件名，所以我们完全可以基于它编写一个程序，搜索 Web、计算网页的文档摘要，并返回一些目标网页，这些网页的摘要与用户查找的页面摘要相似。我们将这个客户程序留给读者作为一个具有挑战性的练习。

这个解决方案仅仅是一个概述。计算机科学家尚在发明和研究许多计算文档摘要并对其进行比较的有效算法。我们目的是向读者介绍这个基本的问题领域，同时阐述解决计算挑战时抽象的强大力量。向量是一种基本的数学抽象，我们可以通过开发抽象层次结构，构建搜索问题的解决方案：由 Java 数组构建 Vector，使用 Vector 构建 Sketch，客户代码使用 Sketch。和往常一样，我们从开发这些 API 的许多尝试来向读者详细介绍这个抽象，但是可以看到，数据类型是根据问题的需要而设计的，同时又考虑了实现的要求。找到正确的抽象设计方案和合理正确的代码实现是有效的面向对象程序设计的关键。抽象的强大可以



通过数学、物理模型和计算机程序的许多例子加以佐证。在后续的任务中，读者会不断开发数据类型以应对自己的计算挑战。你会变得越来越熟练，也将会越来越欣赏这种强大的能力。

程序3.3.5 相似性检测

```
public class CompareDocuments
{
    public static void main(String[] args)
    {
        int k = Integer.parseInt(args[0]);
        int d = Integer.parseInt(args[1]);

        String[] filenames = StdIn.readAllStrings();
        int n = filenames.length;
        Sketch[] a = new Sketch[n];
        for (int i = 0; i < n; i++)
            a[i] = new Sketch(new In(filenames[i]).readAll(), k, d);

        StdOut.print(" ");
        for (int j = 0; j < n; j++)
            StdOut.printf("%8.4s", filenames[j]);
        StdOut.println();
        for (int i = 0; i < n; i++)
        {
            StdOut.printf("%4s", filenames[i]);
            for (int j = 0; j < n; j++)
                StdOut.printf("%8.2f", a[i].similarTo(a[j]));
            StdOut.println();
        }
    }
}
```

k | gram长度
d | 维度
n | 文档数量
a[] | 所有文档摘要数组

这个Sketch客户程序从标准输入中读取文档列表，根据所有文档的 k -gram频率计算文档摘要，并输出所有文档对之间的相似度量表。程序从命令行中接收两个参数： k 的值和维度 d 。

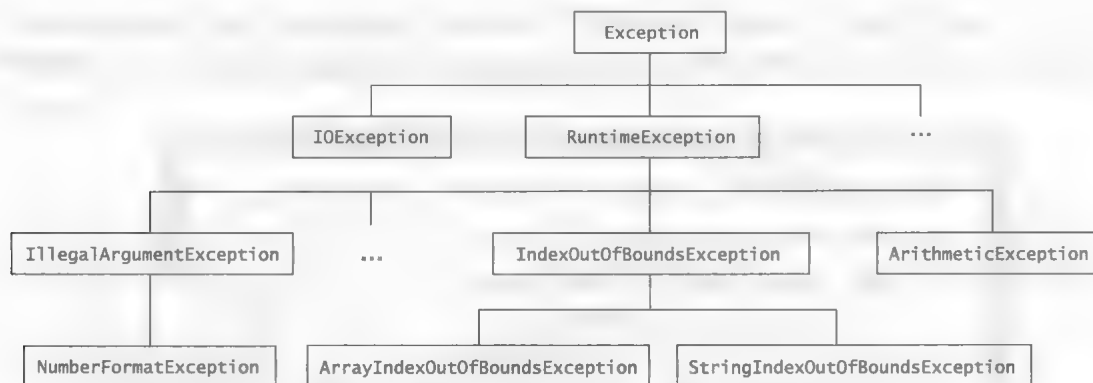
% java CompareDocuments 5 10000 < documents.txt

	Cons	TomS	Huck	Prej	Pict	DJIA	Amaz	ACTG
Cons	1.00	0.66	0.60	0.64	0.20	0.18	0.21	0.11
TomS	0.66	1.00	0.93	0.88	0.12	0.24	0.18	0.14
Huck	0.60	0.93	1.00	0.82	0.09	0.23	0.16	0.12
Prej	0.64	0.88	0.82	1.00	0.11	0.25	0.19	0.15
Pict	0.20	0.12	0.09	0.11	1.00	0.04	0.39	0.03
DJIA	0.18	0.24	0.23	0.25	0.04	1.00	0.16	0.11
Amaz	0.21	0.18	0.16	0.19	0.39	0.16	1.00	0.07
ACTG	0.11	0.14	0.12	0.15	0.03	0.11	0.07	1.00

契约式设计 最后，我们简要讨论一下 Java 语言中的一种机制，它可以用于用户在程序运行时检验某种假设。例如，如果存在一种代表粒子的数据类型，则可能会断言其质量是正的，且速度比光速小。或者，如果存在一个计算两个相同维度向量加法的方法，则可以断言结果向量的维度也保持一致。

异常。异常是程序运行时发生的破坏性事件，通常表示一种错误。相应采取的措施被称为抛出异常。我们在学习程序设计的过程中已经遇到过 Java 标准模块抛出的异常，ArithmeticException、IllegalArgumentException、NumberFormatException 和 ArrayIndexOutOfBoundsException 等都是常见的异常信息。

用户也可以创建并抛出自己的异常。Java 包含一个精心设计的预定义异常的分层继承结构。每个异常类都是 java.lang.Exception 的子类。以下图表显示了这个分层结构的一部分。



465 异常的子类继承层次（部分）

也许最简单的异常是 `RuntimeException`。以下语句创建一个 `RuntimeException`，通常会中断程序的执行并输出一个错误信息。

```
throw new RuntimeException("Custom error message here.");
```

在对用户有帮助的情况下使用异常是一个好习惯。例如，在 `Vector`（程序 3.3.3）中，当用于相加的两个向量维度不一致时，在 `plus()` 中应该抛出一个异常。为此，在 `plus()` 的开头插入以下语句：

```
if (this.coords.length != that.coords.length)
    throw new IllegalArgumentException("Dimensions disagree.");
```

通过这段代码，客户程序会收到 API 违规的精确描述（`Dimensions disagree` 的意思是使用不同维度的向量调用 `plus()` 方法），从而使程序员能够识别错误。如果没有上述代码，`plus()` 方法的结果是无法预计的，可能抛出一个 `ArrayIndexOutOfBoundsException` 或者返回一个不合逻辑的结果，这取决于这两个向量的维数（参见练习 3.3.16）。

断言。断言是一个布尔表达式，用于确定程序执行时的某个特定点应该为 `true`。如果表达式为 `false`，程序将抛出一个 `AssertionError`，通常会终止程序并报告错误信息。错误就像异常，只是错误表示灾难性的失败。`StackOverflowError` 和 `OutOfMemoryError` 是本书以前提到过的两个例子。

程序员广泛使用断言来检测错误并确保程序的正确性。断言也用来记录程序员的意图。例如，在 `Counter`（程序 3.3.2）中，通过在 `increment()` 方法的最后增加如下代码，我们可以检查计数器是否为负数：

```
assert count >= 0;
```

这个声明会捕获计数器值为负的情况。读者还可以添加自定义信息

```
assert count >= 0 : "Negative count detected in increment()";
```

来帮助查找错误。在默认情况下，断言功能是被关闭的，相应语句会被直接忽略，但是可以通过使用 `-enableassertions` 标记（简称为 `-ea`）从命令行启用。断言仅用于调试，正常运行时程序不能依赖于断言（因为断言可能被关闭）。

如果读者选修了系统编程的课程，将会学到如何使用断言来确保代码不会因系统错误而终止或进入无限循环。一种称为契约式设计的模型表述了这种设计理念。数据类型的设计者

规定一个前置条件（调用一个方法时必须满足的条件）、一个后置条件（从方法返回时确保达成的条件）、不变量（运行过程中实现要确保满足的任何条件）以及副作用（方法可能导致的任何其他状态变化）。在开发过程中，可以用断言来测试这些条件。许多程序员使用断言来帮助调试。

本节讨论的语言机制说明了数据类型设计的理念和功效可以帮助我们深入了解编程语言的设计。专家们依旧在争论支持其中一些设计理念的最好方法。为什么 Java 不允许函数作为方法的参数？为什么 Python 不提供强制封装的语言支持？为什么 Matlab 不支持可变数据类型？正如第 1 章所述，与其抱怨一种编程语言的特点，还不如成为一种编程语言的设计者。如果这不是你的计划，那么最佳策略是采用广泛使用的程序设计语言。大多数系统提供大量的库（适当的时候你肯定会采用），但是通过构建抽象，常常可以在简化客户代码的同时确保代码的正确性，并可以轻松转移到其他程序设计语言。你的主要目标是开发数据类型，以便大部分工作适合在问题的抽象层面上完成。

467

问答环节

问：如果尝试访问另一个类的私有实例变量或方法，会发生什么？

答：将得到一个编译时错误，错误提示大意为给定的实例变量或方法在给定的类中具有私有访问权限。

问：Complex 中的实例变量是私有的，但是当使用 `a.plus(b)` 为 Complex 对象执行方法 `plus()` 时，我们不仅可以访问实例变量 `a`，还可以访问 `b`。`b` 的实例变量不应该不能访问吗？

答：私有访问的粒度在类级别而不是实例级别。将实例变量声明为私有意味着其不能从任何其他类直接访问。Complex 类中的方法可以访问（读取或写入）该类中任何对象的实例变量。有一个限制性更强的访问修饰符可能会更好（比如 `superprivate`），这会在实例级别的粒度上限制私有访问，以便只有调用对象才能访问其实例变量，但是 Java 没有这样的功能。

问：Complex（程序 3.3.1）中的 `times()` 方法需要一个构造函数，以极坐标作为参数。如何添加这样的构造函数？

答：我们不能添加这样的构造函数，因为已经有需要两个浮点数作为参数的构造函数。一种更佳方案是在 API 中添加两个常用的函数 `createRect(x, y)` 和 `createPolar(r, theta)`。这种设计方案更好，因为它给客户程序提供了通过指定矩形或极坐标来创建对象的功能。这个例子说明开发数据类型时，考虑多种实现方法是一个不错的主意。

问：本节中定义的 Vector（程序 3.3.3）数据类型与 Java 的 `java.util.Vector` 数据类型之间是否存在关系？

答：不存在关系，只是因为向量确实是线性代数和向量演算的术语，所以我们使用这个名称。

468

问：Vector（程序 3.3.3）中的 `direction()` 方法如果用全零向量调用，程序应该返回什么？

答：一个完整的 API 应该考虑到每种情况下每种方法的行为。在这种情况下，抛出异常或返回 `null` 是适当的。

问：什么是已弃用的方法？

答：一种不再完全支持的方法，但保留在 API 中以保持兼容性。例如，Java 曾经包含一个方法 `Character.isSpace()`，程序员编写依赖于该方法的操作的程序。当 Java 的设计者希望支持更多的 Unicode 空格字符时，他们无法在不破坏客户程序的情况下改变 `isSpace()` 的

行为。为了解决这个问题，设计者添加了一个新的方法 `Character.isWhiteSpace()`，并且弃用旧的方法。随着时间的推移，这种做法肯定会使 API 复杂化。

问：如下 `Complex` 的 `equals()` 的实现有什么错误？

```
public boolean equals(Complex that)
{
    return (this.re == that.re) && (this.im == that.im);
}
```

答：这段代码重载了 `equals()` 方法，而不是重写方法。也就是说，代码定义了一个名为 `equals()` 的新方法，这个方法接收一个 `Complex` 类型的参数。这个重载的方法不同于继承的方法 `equals()`，继承的方法接收一个 `Object` 类型的参数。在一些情况下，如我们在 4.4 节中讨论的 `java.util.HashMap` 库，这个库调用继承的方法而不是重载的方法，因此导致令人费解的行为。

问：如下 `Complex` 的 `hashCode()` 有什么错误？

```
public int hashCode()
{ return -17; }
```

469

答：技术上讲，它满足 `hashCode()` 的约定：如果两个对象相等，则它们具有相同的散列码。然而，由于我们希望 `Math.abs(x.hashCode()%m)` 将 n 个典型 `Complex` 对象分成大小相等的 m 组，因此可能会导致性能不佳。

问：一个接口可以包含构造函数吗？

答：不可以，因为我们不能实例化一个接口，只能实例化实现类的对象。但是，一个接口可以包括常量、方法签名、默认方法、静态方法和嵌套类型，这些功能超出了本书的范围。

问：一个类可以是多个类的直接子类吗？

答：不可以。每个类（`Object` 除外）都是一个且只能是一个父类的直接子类。这个特性被称为单继承。某些其他语言（特别是 C++）支持多继承，一个类可以是两个或更多父类的直接子类。

问：一个类可以实现多个接口吗？

答：可以。为了实现多个接口，请在关键字 `implements` 后列出每个要实现的接口，接口之间由逗号分隔。

问：`Lambda` 表达式的主体可以包含多个单独的语句吗？

答：可以，主体可以是一个语句块，可以包含变量声明、循环和条件。在这种情况下，必须使用显式的 `return` 语句来指定 `Lambda` 表达式的返回值。

问：在某些情况下，`Lambda` 表达式只能在另一个类中调用命名方法。有没有简写形式？

答：有，方法引用是一个紧凑且易读的 `Lambda` 表达式，用于已经具有名称的方法。例如，我们可以使用方法引用 `Gaussian::pdf` 作为 `Lambda` 表达式 `x -> Gaussian.pdf(x)` 的简写。有关详细信息，请参阅本书官网。

470

练习

3.3.1 请使用 `int` 来存储自 1970 年 1 月 1 日以来的秒数，用来表示一个时间点。使用此表示法的程序何时会面对时间爆炸？发生这种情况时该如何处理？

3.3.2 请创建一个数据类型 `Location`，使用球坐标（纬度 / 经度）处理地球上的位置。请在数据类型中包含在地球表面生成随机位置的方法，解析字符串 “25.344 N, 63.5532 W” 得到一个位置，并计算两个位置之间的大圆距离。

3.3.3 请使用 `Counter`（程序 3.3.2）为 `Histogram`（程序 3.2.3）开发一种新的实现。

3.3.4 请使用其他 `Vector` 方法（如 `direction()` 和 `magnitude()`）来为 `Vector` 重新实现一个 `minus()` 方法。
答案：

```
public Vector minus(Vector that)
{ return this.plus(that.scale(-1.0)); }
```

这种实现的优点是限制了要检查的详细代码的数量，缺点是代码可能是低效的。在这种情况下，`plus()` 和 `times()` 都会创建新的 `Vector` 对象，性能会很低。这时如果出于性能的考虑复制 `plus()` 的代码过来并把加号改为减号可能是一种更好的实现方法。

3.3.5 为 `Vector` 实现一个 `main()` 方法来对其方法进行单元测试。

3.3.6 请创建一个数据类型，使用位置（ r_x, r_y, r_z ）、质量（ m ）和速度（ v_x, v_y, v_z ）表示三维空间粒子。数据类型中包含一个返回其动能的方法，动能计算公式为 $1/2m(v_x^2 + v_y^2 + v_z^2)$ 。请使用 `Vector` 数据类型（程序 3.3.3）。

3.3.7 如果读者熟悉物理学，请开发上一道练习中数据类型的另一个版本，使用动量（ p_x, p_y, p_z ）作为实例变量。

471

3.3.8 请实现与 `Vector` 具有相同 API 的二维向量数据类型 `Vector2D`，唯一的区别是其构造函数接收两个 `double` 值作为参数。使用两个 `double` 值（而不是一个数组）作为实例变量。

3.3.9 请实现前面练习中的 `Vector2D` 数据类型，使用一个 `Complex` 对象作为唯一的实例变量。

3.3.10 请证明两个二维单位向量的点积等于其夹角的余弦值。

3.3.11 请实现一个数据类型 `Vector3D`，用于表示三维向量，与 `Vector` 具有相同 API，唯一的区别是其构造函数的参数是三个 `double` 值。另外，增加一个叉积方法：两个向量的叉积的结果是另一个向量，其定义公式如下：

$$\mathbf{a} \times \mathbf{b} = c \|\mathbf{a}\| \|\mathbf{b}\| \sin \theta$$

其中 c 是同时垂直于 \mathbf{a} 和 \mathbf{b} 的单位法向量， θ 是 \mathbf{a} 和 \mathbf{b} 之间的夹角。在直角坐标系中，下面的等式定义了叉积：

$$(a_0, a_1, a_2) \times (b_0, b_1, b_2) = (a_1 b_2 - a_2 b_1, a_2 b_0 - a_0 b_2, a_0 b_1 - a_1 b_0)$$

叉积出现在力矩、角动量和向量算子旋度的定义中。另外， $|\mathbf{a} \times \mathbf{b}|$ 是边长为 \mathbf{a} 和 \mathbf{b} 的平行四边形的面积。

3.3.12 请重写 `Charge`（程序 3.2.6）的 `equals()` 方法，使得两个 `Charge` 对象在具有相同的位置和电荷值时相等。使用本节中描述的 `Objects.hash()` 技术重写 `hashCode()` 方法。

3.3.13 请重载 `Vector`（程序 3.3.3）的 `equals()` 方法和 `hashCode()` 方法，使得两个长度相等且对应的坐标相等的 `Vector` 对象相等。

3.3.14 请向 `Vector` 添加一个 `toString()` 方法，方法返回向量的组成部分，使用逗号分隔，并用括号括起来。

472

3.3.15 请在 `Sketch` 中添加 `toString()` 方法，该方法返回与文档摘要对应的单位向量的字符串表示形式。

3.3.16 如果 x 对应于向量（1,2,3）， y 对应于向量（5,5），描述 `Vector`（程序 3.3.3）中方法调用 `x.add(y)` 和 `y.add(x)` 的行为。

3.3.17 请使用断言和异常来开发一个 `Rational` 的实现，使得程序不受溢出的影响（参见练习 3.2.7）。

3.3.18 请在 `Counter`（程序 3.3.2）中添加代码，如果客户端尝试使用一个负值的 `max` 创建一个 `Counter` 对象，则在运行时抛出 `IllegalArgumentException`。

473

数据类型设计练习

这一组练习题旨在帮助读者获得更多开发数据类型的经验。针对每个问题，设计一个或多个 API 及 API 实现，通过实现典型的客户代码来测试设计方案的正确性。一些练习需要知道特定领域的知识，或者通过 Web 查找相关信息。

3.3.19 统计。请开发一个数据类型，用于维护一组实数的统计信息。提供一种方法来增加数据点，并提供相应的方法以返回点的数量、均值、标准差和方差。开发两种实现：一个实现的实例变量是点的数量、值的和、值的平方和，另一个实现的实例变量则存储包含所有数据点的数组。为了简单起见，构造函数可以指定一个最大数据点数量的参数。第一个实现的速度更快，消耗的内存更少，但更容易出现舍入误差。更好的替代方案可以参考本书官网。

3.3.20 基因组。开发一个数据类型来存储生物体的基因组。生物学家经常将基因组抽象成核苷酸序列（A、C、G 或 T）。数据类型应该支持方法 `addCodon(char c)`、`nucleotideAt(int i)` 以及 `isPotentialGene()`（见程序 3.1.1）。请开发以下三个实现。第一个实现使用一个 `String` 类型的实例变量，使用字符串拼接操作实现 `addCodon()`。每个方法调用花费的时间都与当前基因组长度成比例。第二个实现使用一个字符数组作为实例变量，每当数组被填满时长度会增加一倍。第三个实现使用一个布尔数组，每个碱基使用两位二进制编码，并且每当数组填满时，数组的长度增加一倍。

3.3.21 时间。开发一个表示一天时间的数据类型。请提供返回当前小时、分钟和秒的方法，以及 `toString()`、`equals()` 和 `hashCode()` 方法。开发两种实现：一种将时间存储为一个 `int` 值（自午夜 12 点钟以来的秒数），另一种将时间存储为三个 `int` 值，分别表示秒、分钟和小时。

3.3.22 VIN 号码。请开发数据类型 VIN，针对称为车辆识别号码（VIN）的命名方案。VIN 描述了美国汽车、公共汽车和卡车的品牌、型号、年份和其他属性。

474

3.3.23 生成伪随机数。请开发一个生成伪随机数的数据类型，即将 `StdRandom` 转换为数据类型。不使用 `Math.random()`，数据类型需基于线性同余产生器。这种方法可以追溯到计算机产生初期，也是一个在计算中存储状态值的典型例子。按照如下方案设计一个数据类型：若要生成伪随机 `int` 值，请存储一个 `int` 值 `x`（返回的最后一个“随机”数字的值）。每次客户端请求一个新值时，返回 `a*x+b`，这时我们需要选择适当的 `a` 和 `b` 的值（忽略溢出），使用算术将这些值转换为其他类型数据的“随机”值。D. E. Knuth 建议，`a` 使用值 3141592621，`b` 使用值 2718281829。数据类型需要提供一个构造函数，允许客户程序以一个指定的 `int` 值作为开始（这个 `x` 的初始值被称为种子）。这种功能清楚地表明，产生的数字根本不是随机的（尽管它们可能具有许多随机数的属性），但是这个事实可以用来帮助调试，因为客户程序可以设计成每次都生成相同的数字。

475

创新练习

3.3.24 封装。下面的类是不可变的吗？

```
import java.util.Date;
public class Appointment
{
    private Date date;
    private String contact;

    public Appointment(Date date)
    {
```



```

        this.date = date;
        this.contact = contact;
    }
    public Date getDate()
    { return date; }
}

```

答案：不是。Java 的 `java.util.Date` 类是可变的。方法 `setDate(seconds)` 将调用日期的值更改为自 1970 年 1 月 1 日 00:00:00 GMT 以来的毫秒数。这将产生一个灾难性的后果，当一个客户程序获得 `date=getDate()` 的日期时，客户程序可以调用 `date.setDate()` 并更改 `Appointment` 对象类型中的日期，这可能会产生冲突。在一个数据类型中，不允许可变对象的引用转义，因为调用者可能会修改其状态。这个问题的一种解决方案是在使用 `new Date(date.getTime())` 返回日期之前创建 `Date` 的防御拷贝。当通过 `this.date=new Date(date.getTime())` 存储时，它是一个防御拷贝。很多程序员认为 `Date` 的可变性是 Java 设计的一个缺陷（`GregorianCalendar` 是一个更现代的用于存储日期的 Java 库，但它也是可变的）。

- 3.3.25 日期。开发 Java 的 `java.util.Date` API 的实现，使其不可变，因而可以修正上述练习的缺陷。
- 3.3.26 日历。开发 `Appointment` 和 `Calendar` API，可用于记录日历年中的约会（按天）。目标是让客户程序能够安排不会冲突的约会，并向用户报告当前约会。
- 3.3.27 向量场。向量场将向量与欧几里得空间中的每个点相关联。请编写另一个版本的 `Potential`（练习 3.2.23），从输入接收一个大小为 n 的网格，基于在 $n \times n$ 网格中等距离分布点的点电荷，计算每个点电势向量值，并在每个点绘制电场方向的单位向量（修改 `Charge` 来返回一个 `Vector`）。
- 3.3.28 基因组草图。编写一个函数 `hash()`，以一个 k -gram（长度为 k 的字符串）作为参数，其字符由 A、C、G 或 T 组成，并返回一个介于 0 到 4^k-1 之间的 `int` 值，相当于将字符串转换为四进制（基数为 4）的数字，即将 {A, C, G, T} 分别转换为 {0,1,2,3}。接下来，编写一个函数 `unHash()` 来实现逆变换。使用方法创建一个类似于 `Sketch`（程序 3.3.4）的类 `Genome`，但是类 `Genome` 要基于基因组中 k -grams 的精确计数。最后，为 `Genome` 对象编写 `CompareDocuments`（程序 3.3.5）的一个版本，用于查找本书官网上一组基因组文件之间的相似性。
- 3.3.29 概要分析。从本书官网上选择若干感兴趣的文档（或者使用你自己收集的一组文档），使用多组命令行参数 k 和 d 的值来运行 `CompareDocuments`，以体会参数对计算的影响。
- 3.3.30 多媒体搜索。请开发用于音频和图片的摘要策略，并使用它们来发现自己计算机上音频库中歌曲的相似度和相册中照片之间的相似度。
- 3.3.31 数据挖掘。请编写一个用于浏览 Web 页面的递归程序，从第一个命令行参数指定的页面开始，查找与第二个命令行参数指定的页面相似的页面，处理过程如下：处理名称，打开输入流，执行 `readAll()`，计算文档摘要，如果它与目标页面的距离大于第三个命令行参数所给定的阈值，则输出其名称。然后扫描 `readAll` 得到文字串中所有以“`http://`”字符串开头的页面并（递归地）处理带有这些名称的页面。注意：这个程序可能会读取大量的 Web 页面！

476

477

3.4 案例研究：多体模拟

第 1 章和第 2 章讨论的几个例子可以更清晰地表述面向对象的程序设计。例如，我们可以将 `BouncingBall`（程序 3.1.9）实现为一个值为球的位置和速度的数据类型，客户程序则调用方法来移动和绘制球。这种数据类型为客户程序提供更大能力，如可以同时模拟几个小球的运动（参见练习 3.4.1）。同样，2.4 节中 `Percolation` 的案例研究，以及 1.6 节中随机冲浪的案例研究，很显然都可以作为面向对象程序设计的有趣练习。`Percolation` 将作为练习

3.4.8, 随机冲浪者的面向对象程序设计将在 4.5 节中重新讨论。在本节, 我们将研究一个新的程序来演示面向对象程序设计方法。

我们的任务是编写一个程序, 动态模拟多个物体在相互引力吸引作用下的运动状况。这个多体模拟问题最初由艾萨克·牛顿 (Isaac Newton) 在 350 多年前提出, 至今依然是热门的研究话题。

数据类型应该包含什么值的集合, 有哪些作用于这些值的操作? 本节分析的案例是面向对象程序设计中引人注目的例子, 其原因之一是, 它提出了在真实世界的物理对象与程序中使用的抽象对象之间直接且自然的对应关系。对于许多新手而言, 解决问题的方法从编写执行语句序列转换到设计数据类型是困难的。随着经验的积累, 应用这种方法解决计算问题会使读者从中受益。

首先, 我们回忆一下高中物理学过的一些基本概念和公式。理解程序代码并不需要完全理解这些公式, 由于采用了封装, 这些公式仅仅出现在几个方法中。而且由于采用了数据抽象, 大多数代码非常直观且容易理解。从某种意义上说, 这就是面向对象程序设计的最终目标。

478

多体模拟 1.5 节中的弹跳小球模拟基于牛顿第一运动定律: 任何物体都保持匀速直线运动或静止状态, 直到外力迫使它改变运动状态。通过牛顿第二运动定律 (这解释了外力如何影响运动速度) 改善这个例子, 产生了一个令科学家着迷很长时间的一个基本问题。多体问题即给定 n 个物体的系统 (这些物体通过万有引力相互作用), 如何描述其运动轨迹。同一个基本模型适用于从天体物理学到分子动力学的各种问题。

1687 年, 牛顿在其著作《自然哲学的数学原理》中阐述了两个物体在相互引力作用下运动的原理。但是, 牛顿却无法构建三个物体运动规律的数学模型。研究已经表明, 描述三个物体运动规律的基本方法根本就不存在, 而且在不同的初始条件下, 三个物体的运动行为还可能导致混乱。为了研究此类问题, 科学家不得不求助于开发一个精确的模拟程序。在本节中, 我们开发了一个面向对象的程序来实现这种模拟。科学家热衷于研究数量众多的物体的运动规律, 我们的解决方案仅仅是对这个问题的入门介绍。不过, 结果可能会使你感到惊讶, 因为我们可以开发出逼真的图像来描绘复杂的运动。

Body 数据类型。在 BouncingBall (程序 3.1.9) 中, 我们使用 double 型变量 rx 和 ry 存储小球离原点的距离, 使用 double 型变量 vx 和 vy 存储球的速度, 使用如下语句实现小球在一个时间单位内的位移总量:

```
rx = rx + vx;
ry = ry + vy;
```

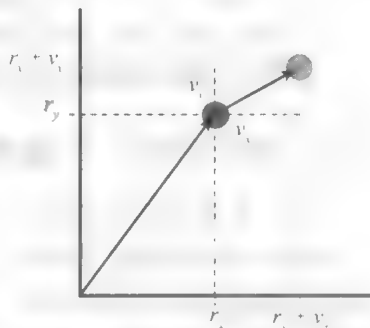
使用 Vector (程序 3.3.3), 我们可以在向量变量 \mathbf{r} 中存储位置, 在向量变量 \mathbf{v} 中存储速度, 然后使用一条语句计算小球在 dt 时间单位内的位移总量:

```
 $\mathbf{r} = \mathbf{r}.\text{plus}(\mathbf{v}.\text{times}(dt));$ 
```

在多体模拟中, 我们将使用类似的操作, 所以首要的设计决策是使用 Vector 对象代替单独的 x 和 y 分量。

479

这个决策能使代码更清晰、紧凑, 并且比单个数据的替代方案更灵活。Body (程序 3.4.1) 是一个 Java 类, 使用



使用 Vector 移动小球

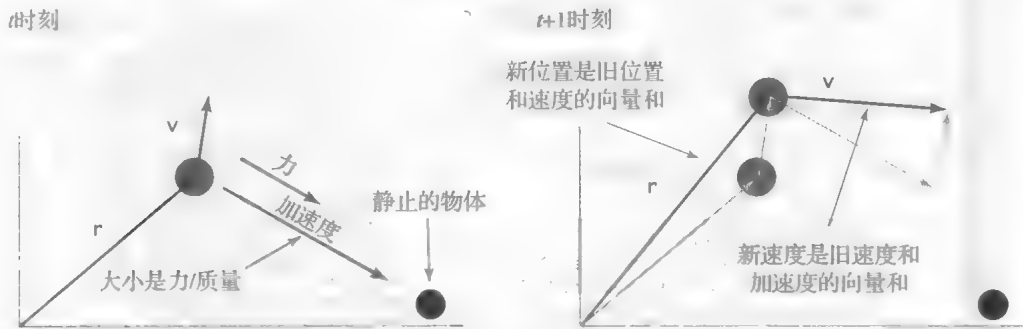
Vector 实现物体移动。该数据类型的实例变量为两个 Vector 对象（分别存储物体的位置和速度），以及一个 double 型变量（存储质量）。数据类型的操作允许客户程序移动和绘制物体（以及计算由其他物体引力产生的力，并用向量表示这个力），Body 类由以下 API 定义：

```
public class Body
{
    Body(Vector r, Vector v, double mass)
    void move(Vector f, double dt)  给对象施加力f，移动对象dt秒
    void draw()                    绘制物体
    Vector forceFrom(Body b)        这个物体在物体b上的外力向量
}
```

基于牛顿定律的多体移动的 API（见程序 3.4.1）

从技术上讲，物体的位置（距原点的位移）不是一个向量（是空间中的一个点，并非方向和大小），但是把它表示为一个向量更加方便，因为向量的操作使移动物体的代码更加紧凑，正如刚刚讨论的那样。当移动物体时，不仅要改变物体的位置，还需要改变其速度。

力与运动。牛顿第二运动定律表明，施加在一个物体（一个向量）上的力等于它的质量（一个向量）和它的加速度（也是一个向量）的标量积： $F=ma$ 。换言之，要计算物体的加速度，可以先计算力，然后用力除以物体的质量。



相对于静止物体的移动

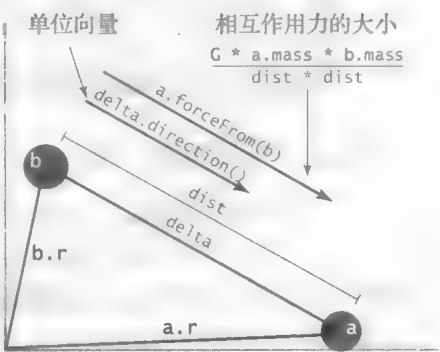
480

在 Body 中，力是方法 move() 的一个参数，其类型为 Vector，因此我们可以先通过力除以质量（存储在 double 型实例变量中）来计算加速度向量，然后通过速度加上其在时间间隔中向量的改变值来计算速度的变化（计算方法与使用速度来改变位置的方法一致）。这个法则可以利用如下代码实现，通过计算给定力向量 f 和时间间隔 dt 来更新物体的位置和速度：

```
Vector a = f.scale(1/mass);
v = v.plus(a.scale(dt));
r = r.plus(v.scale(dt));
```

这段代码包含在 Body 的 move() 方法中，用于调整各值以反映该力在时间间隔内作用于物体的结果：物体移动且速度改变。上述计算假定在时间间隔内加速度保持不变。

物体之间的作用力。Body 的 forceFrom() 中封装了一个物体作用于另一个物体力的计算，以 Body 对象作为参数，并返回一个 Vector 类型。牛顿的万有引力定律是计算的基础：两个物体之间的引力大小等于它们质量



物体之间的作用力

的乘积,除以它们之间距离的平方,再乘以万有引力常数 G (G 为 $6.67 \times 10^{-11} \text{Nm}^2/\text{kg}^2$),引力的方向是两个物体之间的连线。基于万有引力定律,计算 `a.forceFrom(b)` 的代码如下所示:

```
double G = 6.67e-11;
Vector delta = b.r.minus(a.r);
double dist = delta.magnitude();
double magnitude = (G * a.mass * b.mass) / (dist * dist);
Vector force = delta.direction().scale(magnitude);
return force;
```

力向量的大小表示为 `magnitude`,是一个浮点数,力向量的方向与两个物体位置向量差的方向相同。力向量 `force` 就是一个表示其方向的单位向量扩展了 `magnitude` 倍。

程序3.4.1 引力体

```
public class Body
{
    private Vector r;
    private Vector v;
    private final double mass;

    public Body(Vector r0, Vector v0, double m0)
    { r = r0; v = v0; mass = m0; }

    public void move(Vector force, double dt)
    { // 更新位置和速度
      Vector a = force.scale(1/mass);
      v = v.plus(a.scale(dt));
      r = r.plus(v.scale(dt));
    }

    public Vector forceFrom(Body b)
    { // 计算b施加在物体上的力
      Body a = this;
      double G = 6.67e-11;
      Vector delta = b.r.minus(a.r);
      double dist = delta.magnitude();
      double magnitude = (G * a.mass * b.mass) / (dist * dist);
      Vector force = delta.direction().scale(magnitude);
      return force;
    }

    public void draw()
    {
        StdDraw.setPenRadius(0.0125);
        StdDraw.point(r.cartesian(0), r.cartesian(1));
    }
}
```

r	位置
v	速度
mass	质量

force	施加在物体上的作用力
dt	时间增量
a	加速度

a	这个物体
b	另一个物体
G	万有引力常量
delta	从b到a的向量
dist	从b到a的距离
magnitude	力的大小

`Body`数据类型提供了用于模拟物理实体(如行星或原子粒子)运动所需的操作。`Body`是可变类型,其实例变量是物体的位置和速度,通过`move()`方法响应外力作用而改变物体的位置和速度(物体的质量不可变)。`forceFrom()`方法返回一个力向量。

`Universe`数据类型。`Universe`(程序3.4.2)是实现以下API的数据类型:

```
public class Universe
```

<code>Universe(String filename)</code>	根据filename初始化Universe
<code>void increaseTime(double dt)</code>	模拟dt秒的变化
<code>void draw()</code>	绘制Universe

`Universe`类型的API(见程序3.4.2)

Universe 数据类型定义了一个宇宙（其大小、天体的数量，以及一个天体数组）以及两个数据类型操作：`increaseTime()`，用于调整所有天体的位置（以及速度）；`draw()`，用于绘制所有的天体。多体模拟的核心是实现 Universe 中的 `increaseTime()`。计算的第一部分是计算力向量的双重循环结构，力向量描述了每个天体作用于另一个天体的引力。程序应用了叠加原理，即将作用于一个天体的力向量逐个叠加，得到表示所有力的单一向量。在计算所有的力之后，程序调用 `move()` 来为每个天体在固定的时间间隔内应用计算得到的力。

文件格式。按惯例，我们采用数据驱动的设计方法，从一个文件接收输入数据。构造函数从一个文件中读取 Universe 参数和天体描述，文件包含以下信息：

- 天体的数量。
- 宇宙的半径。
- 每个天体的位置、速度和质量。

像往常一样，为了保持一致性，所有的测量都使用标准国际单位（请回顾我们的代码中出现的万有引力常数 G ）。使用上述定义的文件格式，Universe 构造函数的代码十分简单。

```
% more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

% more 3body.txt
3
1.25e11
0.0e00 0.0e00 0.05e04 0.0e00 5.97e24
0.0e00 4.5e10 3.0e04 0.0e00 1.989e30
0.0e00 -4.5e10 -3.0e04 0.0e00 1.989e30

% more 4body.txt
4
5.0e10
-3.5e10 0.0e00 0.0e00 1.4e03 3.0e28
-1.0e10 0.0e00 0.0e00 1.4e04 3.0e28
1.0e10 0.0e00 0.0e00 -1.4e04 3.0e28
3.5e10 0.0e00 0.0e00 -1.4e03 3.0e28
```

← 半径
速度
质量

位置

Universe 文件格式示例

```
public Universe(String filename)
{
    In in = new In(filename);
    n = in.readInt();
    radius = in.readDouble();
    StdDraw.setXscale(-radius, +radius);
    StdDraw.setYscale(-radius, +radius);

    bodies = new Body[n];
    for (int i = 0; i < n; i++)
    {
        double rx = in.readDouble();
        double ry = in.readDouble();
        double[] position = { rx, ry };
        double vx = in.readDouble();
        double vy = in.readDouble();
        double[] velocity = { vx, vy };
        double mass = in.readDouble();
        Vector r = new Vector(position);
        Vector v = new Vector(velocity);
        bodies[i] = new Body(r, v, mass);
    }
}
```

每个天体都由 5 个浮点数描述：天体位置的 x 坐标和 y 坐标、天体初始速度的 x 和 y 分量、天体的质量。

总结一下，我们在 Universe 的测试客户程序 `main()` 中实现了一个数据驱动程序，模拟 n 个天体在相互引力作用下的运动轨迹。构造函数创建一个包含 n 个 Body 对象的数组，从命令行参数指定的文件中读取每个天体的初始位置、初始速度和质量。`increaseTime()` 计算各天体

之间的相互作用力，并基于该信息在时间间隔 dt 之后更新每个天体的加速度、速度和位置。

484 main() 测试程序调用构造函数，然后循环调用 `increaseTime()` 和 `draw()` 来模拟天体的运动。

程序3.4.2 多体模拟

```
public class Universe
{
    private final double radius;
    private final int n;
    private final Body[] bodies;

    public void increaseTime(double dt)
    {
        Vector[] f = new Vector[n];
        for (int i = 0; i < n; i++)
            f[i] = new Vector(new double[2]);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (i != j)
                    f[i] = f[i].plus(bodies[i].forceFrom(bodies[j]));
        for (int i = 0; i < n; i++)
            bodies[i].move(f[i], dt);
    }

    public void draw()
    {
        for (int i = 0; i < n; i++)
            bodies[i].draw();
    }

    public static void main(String[] args)
    {
        Universe newton = new Universe(args[0]);
        double dt = Double.parseDouble(args[1]);
        StdDraw.enableDoubleBuffering();
        while (true)
        {
            StdDraw.clear();
            newton.increaseTime(dt);
            newton.draw();
            StdDraw.show();
            StdDraw.pause(20);
        }
    }
}
```

radius	宇宙的半径
n	天体的数量
bodies[]	多体数组

% java Universe 3body.txt 20000
880 steps



这是一个数据驱动的程序，第一个命令行参数指定了文件，其中的数据定义了 `Universe` 的信息，本程序用于模拟这个宇宙中天体的运动，第二个命令行参数为递增时间间隔。读者可以参阅相关正文内容以了解构造函数的实现。

485

可以在本书官网中找到许多定义各种“宇宙”的文件，我们鼓励你运行这些程序以观察它们的运动轨迹。当观察到这些运动轨迹的时候你就会明白，为什么牛顿难以推导出定义天体运行的轨道公式，即使是少量天体也很难。以下结果图说明了前面给出的数据文件中二体、三体 and 四体实例运行 `Universe` 的轨迹。二体宇宙是一对相互环绕的轨道对，三体宇宙是一颗卫星在两颗行星轨道上跳跃的混乱情况。四体宇宙则是一个相对简单的情况，是两对缓慢旋转的相互环绕的天体对。这些静态图像采用了 `BouncingBall`（程序 3.1.9）中类似的方法，即修改 `Universe` 和 `Body`，在灰色的背景上交替使用白色和黑色绘制天体。运行 `Universal` 时得到的动态图像可以显示天体环绕的真实场景，这通过固定的图片很难辨别。如果基于大量的天体来运行 `Universe`，读者就可以理解为什么模拟对于试图理解复杂问题的科学家来说是一个如此重要的工具了。多体模拟模型是非常通用的，通过尝试不同的输入文

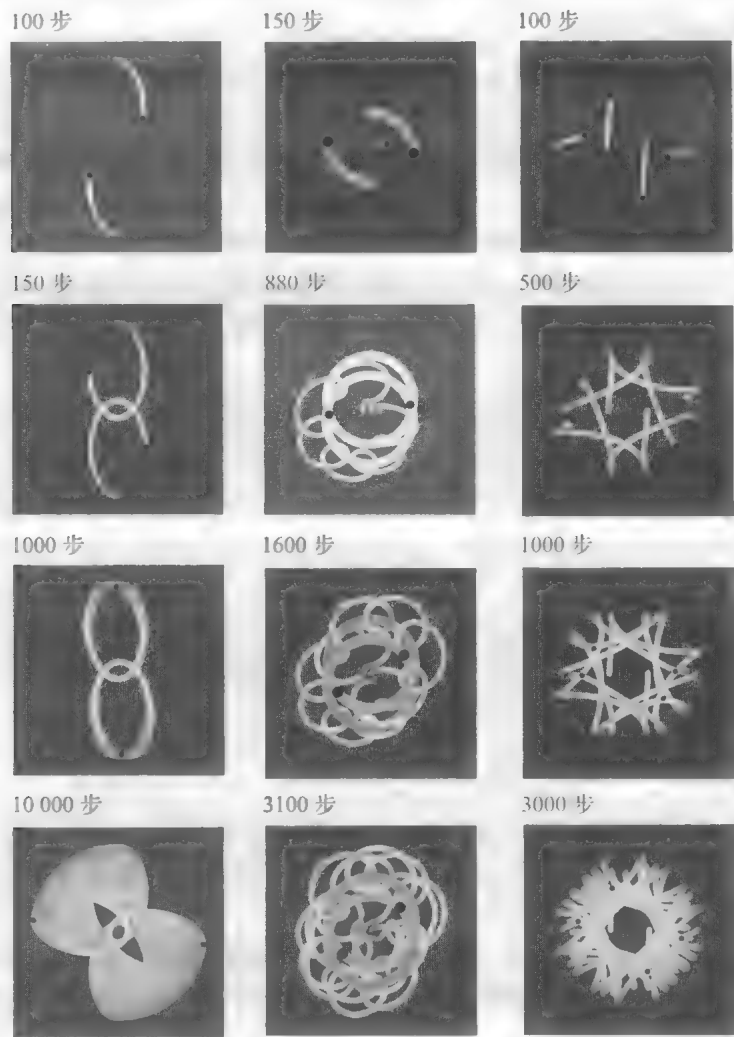
件，读者会充分体会到这一点。

读者可能想要设计自己的 Universe (见练习 3.4.7)。创建一个数据文件的最大挑战是适当地缩放数值，以利用宇宙的半径、时间尺度、天体质量和速度来产生有趣的行为。读者可以研究行星围绕恒星旋转的轨迹，或者亚原子粒子间的相互作用，但是可能无法研究行星与亚原子粒子间的相互作用。当使用自己创建的数据时，很可能会出现一些天体飞到无穷远处、一些会被吸引进入其他天体的情况，但是请欣赏这些情形吧！

```
行星量级
% more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

亚原子量级
% more 2bodyTiny.txt
2
5.0e-10
0.0e00 4.5e-10 1.0e-16 0.0e00 1.5e-30
0.0e00 -4.5e-10 -1.0e-16 0.0e00 1.5e-30
```

486



模拟二体 (左列)、三体 (中部) 和四体 (右列) 宇宙

487

我们提供这个案例的目的是阐述数据类型的效用，而不是提供用于实际用途的多体模拟代码。当使用这种方法研究自然现象时，科学家有许多难题需要处理。第一个是数值精度：在模拟过程中，常常会因为累积计算误差导致模拟产生奇怪的结果，这在自然界中是不会发生的。例如，当天体 (几乎) 相互碰撞时，我们的代码不会采取特殊的行动。第二个是效率：

Universe 中的 `move()` 方法执行的时间与 n^2 成正比，所以不适用于模拟大量的天体。与基因组学一样，要解决与多体相关的科学问题，不仅需要涉及原始问题领域的知识，还需要涉及一些计算机科学家早期就开始研究的核心计算问题。

为了简单起见，我们研究的是一个二维宇宙空间，只有天体在一个平面上运动时，二维宇宙才有现实意义。但是，基于 `Vector` 实现的 `Body` 意味着，无须修改太多代码，一个客户端就可以使用三维向量来模拟三维空间（实际上是任意数量的维度）中球体的运动轨迹！对于三维宇宙，`draw()` 方法可以将球体位置投影到由前两个维度定义的平面上。

Universe 中的测试客户程序只是一种可能性，我们可以在其他各种情况下使用相同的基本模型（例如，在天体间包含不同类型的相互作用力）。一种可能的应用场景是观察和测量一些现有天体的当前运动，然后逆向运行模拟！这是天体物理学家试图了解宇宙起源的一种方法。在科学研究中，我们试图了解事物的过去和预测事物的未来。通过一个良好的模拟，我们可以同时实现这两个目的。

488

问答环节

问：Universe 的 API 规模比较小。为什么不在 `Body` 的测试客户程序 `main()` 中实现这些代码？

答：我们的设计理念是大多数人对宇宙认知的一个表达：宇宙被创造，并随时间而运动。这种设计方法可以使代码更加清晰，同时也能提供最大的灵活性来模拟宇宙是如何运行的。

问：为什么 `forceFrom()` 是一个实例方法？它被实现为采用两个 `Body` 对象作为参数的函数是否更加恰当？

答：是的，`forceFrom()` 的实例方法实现有几种可能的替代方法，其中之一是静态方法，包含两个 `Body` 对象作为参数是合理的选择。一些程序员倾向于在数据类型实现中避免使用静态函数。另一种选择是将作用于每个 `Body` 对象的力作为一个实例变量。我们的选择是两者的折中。

489

练习

3.4.1 开发 `BouncingBall`（程序 3.1.9）的一个面向对象的版本。程序包括：一个构造函数，用于初始化每个小球的运动（随机方向和随机速度，在合理的范围内）；一个测试客户程序，从命令行接收一个整数参数 n ，模拟 n 个弹跳小球。

3.4.2 请在程序 3.4.1 中添加 `main()` 方法，对 `Body` 数据类型对象进行单元测试。

3.4.3 修改 `Body`（程序 3.4.1），实现绘制的圆（对应于天体）半径与其质量成正比。

3.4.4 请问在一个没有引力的宇宙中会发生什么？这种情况对应于 `Body` 中的 `forceTo()` 总是返回零向量。

3.4.5 请创建一个数据类型 `Universe3D` 以模拟三维宇宙。开发一个数据文件用于模拟太阳系中行星围绕太阳旋转的运动轨迹。

3.4.6 请编写一个 `RandomBody` 类，使用（精心选择的）随机值代替参数初始化其实例变量。然后编写一个客户程序，从命令行接收一个整数参数 n ，模拟一个包含 n 个天体的宇宙中天体随机运动的轨迹。

490

创新练习

3.4.7 新宇宙。请使用有趣的属性设计一个新宇宙，并使用 `Universe` 模拟其运动轨迹。这个练习是一个锻炼创造性的好机会！

3.4.8 渗透原理。请开发 `Percolation`（程序 2.4.5）的一个面向对象版本。程序编写之前请仔细考虑设计方案，并证明该设计方案的正确性。

491
492

算法和数据结构

本章讨论基本数据类型，它们是应用程序的基本构件。无论读者选择使用 Java 库实现，还是根据本章给定的代码开发自定义类型，在本章你都会学到如何用好这些数据类型。

对象可以包含指向其他对象的引用，所以我们构建了链接结构，这个数据结构可能会非常复杂。基于链接结构和数组，我们可以构建数据结构来组织信息，以便使用相关算法高效处理数据。在一个数据类型中，我们使用一系列值来构建数据结构，使用方法操作这些值来实现算法。

本章讨论的算法和数据结构知识是过去 50 年来形成的知识体系，它们构成了在各种各样的应用中有效使用计算机的基础。从物理学上的多体模拟问题到生物信息学中的遗传序列问题，从数据库系统到搜索引擎，这些方法都是商业计算的基础。随着计算应用程序范围的不断扩大，这些基本方法的影响也在不断增长。

算法和数据结构本身也是科学研究的对象。因此，我们首先讨论分析算法性能的科学方法，然后在本章的后续章节中，我们会使用这种方法来研究算法的性能。

493

4.1 性能

在本节中，我们将反复强调在编写任何程序时都必须遵守的一个原则，该原则可以简洁地表述为一个设计理念：关注成本。如果你是一位工程师，那么节省成本就是你的工作；如果你是一个生物学家或物理学家，成本将决定你可以解决哪些科学问题；如果你从事商务工作或是一位经济学家，那么节约成本是毋庸置疑的；如果你是一名软件开发人员，成本将决定你构建的软件是否对客户有用。

为了研究软件的运行成本，我们通过科学方法（也就是科学家普遍接受和使用，以研究自然世界知识的技术）来研究程序。我们还将使用数学分析来推导出有关成本的简洁模型。

我们在研究自然界的哪些特征？在大多数情况下，我们对其中一个基本特征感兴趣：时间。每当运行一个程序时，都执行了一次涉及自然界的实验，一个复杂的电子电路系统通过一系列的状态改变，处理大量的离散事件并最终停留在一种稳定状态，以表示我们期望的结果。虽然开发 Java 程序是在抽象世界中进行的，但这些事情肯定会在自然界发生。从开始到我们能够看见结果一共会经过多少时间？对于我们而言，时间的长短（毫秒、秒、天还是星期）十分关键。因此，我们需要利用科学的方法来掌握如何合理控制这些状况，就像发射火箭、建造桥梁或是粉碎原子一样。

一方面，现代程序和编程环境十分复杂；另一方面，它们提供了一套简单（但功能强大）的抽象集以用于程序开发，每次运行程序都可以产生相同的结果。不能不说这是一个小小的奇迹。为了估算程序运行所需的时间，我们利用了构建程序的支撑架构。你可能想象不到编写成本开销估计和性能预测的程序的简单程度。

494

科学方法。以下五个步骤简要总结了我们采用的科学方法：

程序4.1.1 三数求和问题

```
public class ThreeSum
{
    public static void printTriples(int[] a)
    {
        /*参见练习4.1.1*/
    }
    public static int countTriples(int[] a)
    {
        // 计算和为0的三元组

        int n = a.length;
        int count = 0;
        for (int i = 0; i < n; i++)
            for (int j = i+1; j < n; j++)
                for (int k = j+1; k < n; k++)
                    if (a[i] + a[j] + a[k] == 0)
                        count++;
        return count;
    }
    public static void main(String[] args)
    {
        int[] a = StdIn.readAllInts();
        int count = countTriples(a);
        StdOut.println(count);
        if (count < 10) printTriples(a);
    }
}
```

n	数组的长度
a[]	n个整数的数组
count	三个数和为0的个数

countTriples()方法计算a[]中三个数的总和正好为0的三元组（忽略整数溢出）。测试客户程序从标准输入读取一个整数数组，调用countTriples()，如果个数比较少，则同时输出满足条件的三元组。文件1Kints.txt包含1024个随机整数值。这个文件不太可能包含这样的三元组（参见练习4.1.28）。

% more 8ints.txt

```
30
-30
-20
-10
40
0
10
5
```

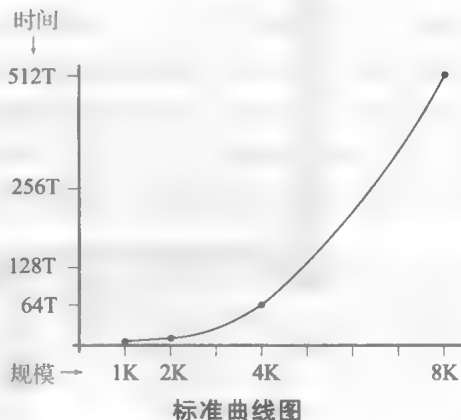
% java ThreeSum < 8ints.txt

```
4
30 -30 0
30 -20 -10
-30 -10 40
-10 0 10
```

% java ThreeSum < 1Kints.txt

```
0
```

实证分析。显然，我们可以通过将输入规模加倍，并观察其对运行时间的影响，来得到一个2倍的倍增假设的答案。例如，DoublingTest（程序4.1.2）为ThreeSum生成一系列随机输入数组，每一步数组长度加倍，然后输出ThreeSum.countTriples()的运行时间相对于上一次运行时间（前一次输入的大小是当前输入大小的一半）的比值。如果运行这个程序，你将发现自己陷入了一种“预测-验证”循环：开始几行的输出速度很快，但随后开始变慢。每次输出一行内容，你将发现自己总会思考这个问题：输出下一行的内容需要多长时间？如果程序运行时使用Stopwatch执行测量，很容易预测每一行的间隔时间按8的倍数增加。因而可以直接推导出一种假设：当输入的大小加倍时，运行时间按8的倍数增加。我们也可以绘制运行时间图，从标准曲线图（右图）可以看出运行时间随着输入规模的增加而增加的比例。也可以基于对数图



496
497

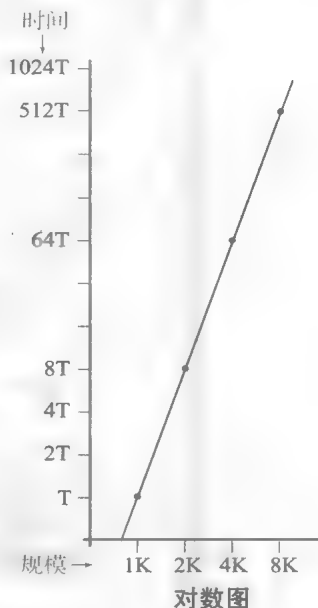
来绘制。针对 ThreeSum 程序，其对数图（下图）是斜率为 3 的直线，该直线充分地表明运行时间满足形如 cn^3 的乘幂定律的假设（见练习 4.1.6）。

数学分析。Knuth 提出的通过构建数学模型来描述程序运行时间的见解十分简单，即程序的总时间取决于以下两个主要因素：

- 每条语句的执行时间成本。
- 每条语句的执行频率。

第一个因素是系统属性，第二个因素是算法属性。如果我们了解程序中所有指令的这两种属性，那么可以通过将属性相乘并加上程序中所有指令的执行时间，来获得程序的运行时间成本。

其中最大的挑战是确定语句的执行频率。有些语句比较容易分析，如 ThreeSum.countTriples() 中初始化 count 为 0 的语句只执行一次。其他语句则需要更高层次的推理，如 ThreeSum.countTriples() 中的 if 语句精确地执行了 $n(n-1)(n-2)/6$ 次（这正是从输入数组中挑选三个不同数的方法的数量——参见练习 4.1.4）。



498

程序4.1.2 验证倍增假设

```
public class DoublingTest
{
    public static double timeTrial(int n)
    { // 计算解决问题的时间，问题的输入大小为n
        int[] a = new int[n];
        for (int i = 0; i < n; i++)
            a[i] = StdRandom.uniform(2000000) - 1000000;
        Stopwatch timer = new Stopwatch();
        int count = ThreeSum.countTriples(a);
        return timer.elapsedTime();
    }

    public static void main(String[] args)
    { // 输出倍增比例列表
        for (int n = 512; true; n *= 2)
        { // 输出问题规模为n的倍增比例
            double previous = timeTrial(n/2);
            double current = timeTrial(n);
            double ratio = current / previous;
            StdOut.printf("%7d %4.2f\n", n, ratio);
        }
    }
}
```

n	问题规模
a[]	随机整数
timer	秒表

n	问题规模
previous	n/2的运行时间
current	n的运行时间
ratio	运行时间比例

该程序向标准输出写入三数求和问题的倍增比例列表。列表显示了倍增问题规模对函数调用 ThreeSum.countTriples() 的运行时间的影响，问题规模初始为 512。列表中每一行表示的问题规模都比上一行加倍。这些实验可以推导出一个假设，即当输入规模加倍时，运行时间按 8 的倍数递增。运行程序时，请注意观察每行内容输出的时间间隔按 8 的倍数递增，从而验证该假设。

```
% java DoublingTest
512 6.48
1024 8.30
2048 7.75
4096 8.00
8192 8.05
...
```

499

```

public class ThreeSum
{
    public static int count(int[] a)
    {
        int n = a.length;
        int count = 0;
        for (int i = 0; i < n; i++)
        {
            for (int j = i+1; j < n; j++)
            {
                for (int k = j+1; k < n; k++)
                {
                    if (a[i] + a[j] + a[k] == 0)
                        count++;
                }
            }
        }
        return count;
    }

    public static void main(String[] args)
    {
        int[] a = StdIn.readAllInts();
        int count = count(a);
        StdOut.println(count);
    }
}

```

内层循环

依赖输入数据

1

n

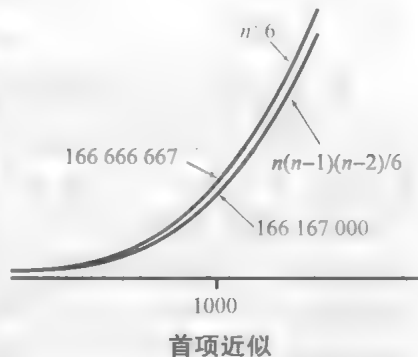
$n/2$

$n/6$

程序语句执行频率的分析

这种类型的频率分析可能导致复杂和冗长的数学表达式。为了显著地简化数学分析中的问题，我们建立了两种更简单的近似表达式。

首先，我们使用称为波浪线表示法的数学方法来处理数学表达式的首项。我们使用记号 $\sim f(n)$ 表示当 n 增大时，除以 $f(n)$ 后趋于 1 的任意量。同样我们使用记号 $g(n) \sim f(n)$ 表示当 n 增加时， $g(n)/f(n)$ 趋于 1。使用这种记法，我们可以忽略一个表达式中代表较小值的复杂部分。例如，ThreeSum 中的 if 语句被执行 $\sim n^3/6$ 次，因为 $n(n-1)(n-2)/6 = n^3/6 - n^2/2 + n/3$ ，如果除以 $n^3/6$ ，则当 n 增大时，结果趋于 1。这种记法适用于首项之后的项相对来说不太重要的情况（例如，当 $n=1000$ 时，这个假设是指与 $n^3/6 \approx 166666667$ 相比， $-n^2/2 + n/3 \approx -499667$ 是微不足道的，结果正是如此）。



其次，我们关注最经常执行的指令，有时是指程序最里层的循环。在这个程序中，我们可以合理地假设内层循环外的指令消耗的时间相对较少。

分析一个程序的运行时间的关键点在于，对于大多数程序，运行时间满足如下关系式：

$$T(n) \sim cf(n)$$

其中 c 是一个常数， $f(n)$ 是一个函数，称为运行时间的增长量级。对于典型的程序， $f(n)$ 是类似于 $\log n$ 、 n 、 $n \log n$ 、 n^2 、 n^3 的函数，我们接下来将接触到这些函数（通常表述增长量级函数时忽略其常数系数）。当 $f(n)$ 是 n 的乘幂时（大多数情况），这个假设等效于运行时间满足乘幂定律。以 ThreeSum 为例，该假设已被我们的经验观察验证：ThreeSum 运行时间的增长量级为 n^3 。常数 c 的值取决于执行语句的时间成本和频率分析的细节，但是一般我们不需要考虑这个常量值，稍后将说明理由。

增长量级是一个关于运行时间的简单而强大的模型。例如，根据增长量级能够自然而然地引出倍增假设。以 ThreeSum 为例，已知增长量级为 n^3 ，表明当问题规模翻倍时，可以估计运行时间将增加 8 倍，因为

$$T(2n)/T(n) = c(2n)^3/cn^3 = 8$$

计算结果与经验分析的结果一致，从而同时验证了模型和实验结果。请仔细研究此示例，因为可以使用相同的方法更好地理解你所编写的任何一个程序的性能。

Knuth 证明了可以为任何一个程序的运行时间建立一个精确的数学模型，并且许多专家花费了大量时间开发这类模型。但是理解一个程序的性能并不需要这样详细的模型：在运行时间估计中，忽略内层循环以外指令的运行时间成本通常是安全的（因为与内层循环中指令的时间成本相比，其运行成本可以忽略不计），且不必知道近似公式中常数的值（因为使用倍增假说进行预测时，常量的作用将被抵消）。

指令数量	每条指令的执行 时间（单位为秒）	指令执行频率	总时间
6	2×10^{-9}	$n^3/6 - n^2/2 + n/3$	$(2n^3 - 6n^2 + 4n) \times 10^{-9}$
4	3×10^{-9}	$n^2/2 - n/2$	$(6n^2 + 6n) \times 10^{-9}$
4	3×10^{-9}	n	$(12n) \times 10^{-9}$
10	1×10^{-9}	1	10×10^{-9}
		全部时间	$(2n^3 + 22n + 10) \times 10^{-9}$
		波浪符	$\sim 2n^3 \times 10^{-9}$
		增长量级	n^3

分析程序的运行时间（示例）

这些近似使得具体使用的机器的特性在模型中不再起到重要作用——这种分析方法把算法从系统中分离开。ThreeSum 程序运行时间的增长量级为 n^3 ，并且不依赖于是在 Java 还是在 Python 中实现的，也不依赖于是在笔记本电脑、手机，还是在超级计算机上运行的。其主要取决于程序检测所有的三元组的功能。计算机和系统的属性均体现在程序语句和机器指令的关系上，表现为我们观测到的程序的实际运行时间，这些都是倍增假设的基础。我们使用的算法决定了增长的量级。这种分离的思想是一种强大的概念，因为它允许我们形成关于算法性能的知识，然后把这些知识应用到任何计算机上。事实上，大多数经典算法性能的研究在几十年前已经完成，但这些知识仍与当今的计算机息息相关。

上述经验和数学分析构成了一个模型，可以通过列出所有相关的假设（例如，每条指令每次执行时消耗同样的时间、运行环境总是一致的等）

为参数，实现对程序执行过程的形式化分析。大多数程序并不值得详细建模，但是读者应该了解自己编写的每个程序的期望运行时间。一定要注意程序的开销。倍增假设是一个很实用的分析方法，你可以通过实验分析，也可以使用数学分析，当然最好是两者结合。关于性能的信息非常重要，你会很快习惯于每次运行一个程序时先对性能有一个假设估算，然后用运行结果验证你的假设。事实上，当你等待程序运行结束时，进行这项工作是对时间的充分利用！

增长量级分类 我们仅仅使用若干结构化原语（语句、选择结构、循环结构和函数调用）就可以构建 Java 程序，所以程序的增长量级通常是程序规模

增长量级		
描述	函数	倍增假设的倍数
常量	1	1
对数	$\log n$	1
线性	n	2
线性对数	$n \log n$	2
二次	n^2	4
三次	n^3	8
指数	2^n	2^n

常见的增长量级分类

的函数，这些函数的类型并不多，我们在上表中总结了一些。看到这些函数你自然会想到倍增假设，我们可以通过实际运行程序来验证这些函数描述是否准确。事实上，你已经运行过一些具有这样的增长量级的程序，接下来我们会简单讨论。

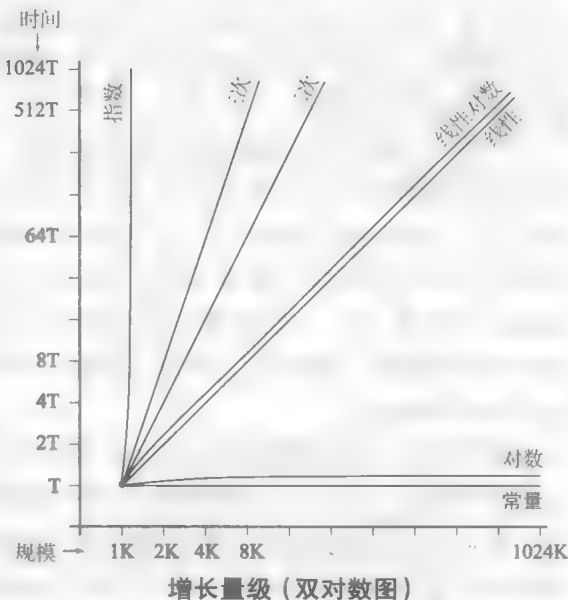
常量型。运行时间的增长量级为常数的程序通常执行固定数量的语句来完成任务。因而其运行时间不依赖于问题规模。第 1 章中的前几个程序（如 HelloWorld（程序 1.1.1）和 LeapYear（程序 1.2.4））就属于这一类。Java 中所有对基本类型的操作需要的时间都是常数级的，Java 的 Math 库中函数消耗的时间同样是常量。请注意，我们说的常量是指不随着输入数据变化，并不是指每个函数的运行时间都相同。例如，Math.tan() 消耗的时间常量比 Math.abs() 消耗的要大。

对数型。运行时间的增长量级为对数的程序比常量时间的程序稍慢。运行时间为问题规模对数的经典程序示例是在一个有序数组中查找一个值，我们将在下一节中讨论这个问题（参见程序 4.2.3 中的 BinarySearch）。对数的底数与增长量级无关（因为底数为任意常数的所有对数都可以通过一个常量因子相互换算），所以我们使用 $\log n$ 表示其增长量级。当我们关心首项中的常数参数（如使用波浪符号）时，需要谨慎指定对数的基数。如符号 $\lg n$ 表示二进制（底数为 2）对数， $\ln n$ 表示自然（底数为 e）对数。

线性型。很多程序处理每条输入数据或者单个 for 循环结构所消耗的时间为一个常量。这类程序的增长量级为线性的——其运行时间直接与问题规模成正比。用于计算标准输入中数值平均值的程序 Average（程序 1.5.3）就是一个典型的例子，在 1.4 节中混排一个数组中元素的代码也是一个例子。像 PlotFilter（程序 1.5.5）过滤器程序也属于这一分类，还有在 3.2 节中讨论的各种图像处理过滤器，它们针对每一个输入像素执行固定数量的数学运算。

线性对数型。对于问题规模 n ，如果程序运行时间的增长量级为 $n \log n$ ，则使用术语线性对数描述该程序。同样，程序运行时间的增长量级与对数的底数无关。例如，CouponCollector（程序 1.4.2）就属于线性对数型，另一个典型的例子是归并排序（见程序 4.2.6）。有一些重要问题的直接解决方案是二次型的，但通过巧妙的算法设计可以实现为线性对数类型。这类算法（包括归并排序）具有非常重要的实际应用价值，使用它们能够解决比二次型解决方案可解决的规模更大的问题。在 4.2 节中，我们将讨论一种用于开发线性对数算法的通用设计技巧，称为分治算法。

二次型。一个运行时间的增长量级为 n^2 的典型程序一般包括两个嵌套的 for 循环，用于某种处理所有 n 个元素两两组合形成的数对的计算。这种类型程序的典型例子是 Universe（程序 3.4.2）中计算任意两个星球之间作用力的双重嵌套循环。我们在 4.2 节中讨论的插入排序算法（程序 4.2.4）也是一个典型的例子。



503

504

名称	增长量级	示例	结构
常量	1	<code>count++;</code>	语句 (整数递增)
对数	$\log n$	<code>for (int i = n; i > 0; i /= 2) count++;</code>	划分成两半 (二进制表示中的位)
线性	n	<code>for (int i = 0; i < n; i++) if (a[i] == 0) count++;</code>	单层循环 (检查每个元素)
线性对数	$n \log n$	[见 <i>Mergesort</i> (程序4.2.6)]	分治 (归并排序)
二次	n^2	<code>for (int i = 0; i < n; i++) for (int j = i+1; j < n; j++) if (a[i] + a[j] == 0) count++;</code>	双层嵌套循环 (检查元素对)
三次	n^3	<code>for (int i = 0; i < n; i++) for (int j = i+1; j < n; j++) for (int k = j+1; k < n; k++) if (a[i] + a[j] + a[k] == 0) count++;</code>	三层嵌套循环 (检查所有三元组)
指数	2^n	[see <i>Gray code</i> (PROGRAM 2.3.3)]	穷举搜索 (检查所有子集)

常见增长量级假设

三次型。本章的例子 *ThreeSum* 属于三次型 (其运行时间增长量级为 n^3)，因为程序包括 3 个嵌套的 `for` 循环，用来处理 n 个元素的所有三元组。如 1.4 节所述，用于两个 $m \times m$ 矩阵的相乘运算的运行时间的增长量级为 m^3 ，所以基本的矩阵乘法算法通常为三次型。然而，输入的大小 (矩阵元素的个数) 与 $n=m^3$ 成正比，所以算法最好归类为 $n^{3/2}$ ，而不是三次型。

指数型。正如 2.3 节所述，*TowersOfHanoi* (程序 2.3.2) 和 *Beckett* (程序 2.3.3) 的运行时间都与 2^n 成正比，因为程序需要处理 n 个元素的所有子集。通常，如果一个程序的增长量级可以描述为 $2^{a \times n^b}$ ，那么对于任何正常数 a 和 b ，我们都把它称为指数型算法，即使不同的 a 和 b 值会导致运行时间大不相同。指数型算法非常慢，通常不建议对于大规模问题运行此类程序。指数型算法在算法理论中占据重要地位，因为存在一大类问题，似乎指数时间的算法是其最佳选择。

本节中列出的增长量级分类是最常用的，但显然不是最全面的。事实上，算法的详细分析可能需要数百年来开发的全部数学工具。理解诸如 *Factors* (程序 1.3.9)、*PrimeSieve* (程序 1.4.3) 和 *Euclid* (程序 2.3.1) 等程序的运行时间需要数论方面的基本结论。诸如 *HashST* (程序 4.4.3) 和 *BST* (程序 4.4.4) 经典算法要求细致的数学分析。*Sqrt* (程序 1.3.6) 和 *Markov* (程序 1.6.3) 是数值计算的原型：它们的运行时间取决于计算到期望数值的收敛率。而我们感兴趣的一些问题，诸如 *Gambler* (程序 1.3.8) 及其变体之类的模拟，其详细数学模型根本就不存在。

尽管如此，你编写的很多程序具有直观的性能特征，可以使用我们描述的某种增长量级进行精确描述。因此，通常我们可以使用高层次的假设来描述一个算法的运行时间特性。例如，我们可以说归并排序运行时间的增长量级是线性对数型。为了方便，我们把这种陈述简化成归并排序是线性对数型算法。我们大多数关于运行时间成本的描述都使用这种形式，或者采用一种比较式的描述，如归并排序比插入排序更快。同样，这些描述的一个显著特征是

它们不是关于程序特性的描述，而是关于算法的陈述。506

预测 我们可以通过简单地运行程序来估计程序的运行时间，但是当问题规模很大时，这并不是一个好的方法。这种情况类似于为了预测火箭的落地点而发射火箭，为了了解炸弹的破坏力而引爆炸弹，或者为了预测桥梁是否坚固而建设桥梁。

了解运行时间的增长量级，可以帮助我们在解决大规模问题时做出决策，以便投入合适的资源来处理实际需要解决的具体问题。通常使用下列方式之一来验证程序运行时间增长量级的假设。

估计解决大型问题的可行性。为了关注运行成本，对于编写的每个程序，你都需要回答这样一个基本问题：程序能够在合理的时间内处理其输入的数据吗？例如，如果问题规模为 n ，三次型算法需要运行几秒，当问题规模为 $100n$ 时，程序将需要几周的时间，因为其会慢 100 万倍，而几百万秒的时间就是几个星期。如果你需要解决的问题规模确实如此，则必须找到一个更好的解决方法。了解算法运行时间的增长量级，可以准确地得到程序可解决的问题规模极限的信息。这就是研究性能的最主要的原因。如果没有这些信息，我们可能无法知道一个程序将消耗多少时间；通过这些信息，则可以通过粗略计算来估计运行成本。

增长量级	问题规模增加100倍后 预期的运行时间
线性	几分钟
线性对数	几分钟
二次	几小时
三次	几周
指数	无穷大

对运行几秒的程序增加问题规模产生的影响

评估使用更快计算机的价值。要关注运行成本，还面临这样一个基本问题：如果计算机速度更快，那么可以以多快的速度解决这个问题？同样，了解运行时间的增长量级正好提供了我们所需要的信息。一个称为摩尔定律的著名经验法则表明，每隔 18 个月，计算机的运行速度可以提高 1 倍，内存可以增加 1 倍；或者每隔 5 年，计算机的速度可以提高 10 倍，内存容量增加 10 倍。一般很自然地认为如果我们购买的新计算机速度提高了 10 倍，而且内存容量比旧计算机容量增加 10 倍，那么我们可以解决 10 倍规模的问题，但实际上，对于二次型或三次型算法，结论并非如此。无论是投资银行家每天运行的财务模型，还是科学家运行程序以分析实验数据，或者是工程师运行模拟以测试一种设计方案，通常都需要几小时才能运行完成程序。假设我们正在运行一个时间为三次型的程序，然后购买一台速度快 10 倍、内存大 10 倍的新计算机，不仅是因为需要一台新的计算机，而是因为你希望处理的问题规模也能增大 10 倍。但是让人失望的是，我们很容易就能预测到新问题需要几个星期才能得出结果，因为新问题增大了 1000 倍，而新计算机的速度比起旧计算机仅仅提高了 10 倍。这种情况就凸显了线性和线性对数算法的重要价值：对于线性和线性对数算法，对于一台比旧计算机快 10 倍、内存容量大 10 倍的新计算机，可以在同样的时间内解决比旧计算机能解决的问题规模大 10 倍的问题。换句话说，如果使用二次型或三次型算法，则根本无法跟上摩尔定律的速度。

增长量级	运行时间增加的倍数
线性	1
线性对数	1
二次	10
三次	100
指数	无穷大

使用快 10 倍的计算机处理大 10 倍的问题的效果比较

比较程序。我们一直在努力改进程序，也经常扩展或修正假设以评估改进的有效性。基

于对性能的预测，在开发过程中可以进行设计决策来指导我们实现更好、更高效的代码。举例来说，新手程序员可能已经在 ThreeSum（程序 4.1.1）中编写了嵌套 for 循环，如下所示：

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            if (i < j && j < k)
                if (a[i] + a[j] + a[k] == 0)
                    count++;
```

这段代码中内层循环中指令的执行频率恰好是 n^3 （而不是约为 $n^3/6$ ）。制定和验证“这个变体程序比 ThreeSum 慢 6 倍”的假设是很容易的。值得注意的是，对于不在内层循环中的代码，这样的改进带来的影响微乎其微。

更一般地说，如果给定两个解决相同问题的算法，我们想知道哪个算法解决问题时使用更少的计算资源。在很多情况下，我们可以确定运行时间的增长量级，建立关于比较性能的准确假设。增长量级在这个过程中十分重要，因为它允许我们把一个特定的算法与其他所有类别的算法进行比较。例如，如果一个问题已经有线性对数算法解决方案，则我们对由二次型或三次型算法（即使它们被高度优化）解决同样的问题就不太感兴趣了。

注意事项 当尝试详细分析程序性能时，许多原因都可能导致不一致或错误的结果。所有的原因或多或少都与“我们的结论中一个或多个基本假设不太正确”有关。我们可以基于新的假设来得到新的结论。在分析的过程中，我们考虑的细节越多，越要认真仔细地分析。

指令时间。我们假设每条指令总是花费相同的时间，其实这并不总是正确的。例如，大多数现代计算机系统均使用一种称为缓存（cache）的技术来管理内存，在这种情况下，如果数组中的数字位置并不相邻，访问超大数组中的元素可能会花费较长的时间。通过让 DoublingTest 运行一段时间，可以观察到缓存对 ThreeSum 的影响。你会发现在收敛到 8 之后，如果继续增大数据规模，运行时间的比值将跳跃到一个较大的数值，这是缓存对程序行为的影响造成的。

非主导地位的内层循环。内层循环占主导地位的假设可能并不总是正确的。问题规模 n 也许不够大，内层循环指令的执行频率虽然是主导项，但相对于次要项的差异没有大到可以忽略次要项。有些程序在内层循环之外包含大量代码，也需要加以考虑。

系统考虑。通常情况下，计算机中有许多程序在运行。Java 仅仅是许多竞争资源的应用程序之一，而 Java 本身包含很多可以显著影响性能的控制选项。这些可能会干扰科学方法中的基本原则：实验应该是可以复现的，但是这个时刻发生在计算机上的事情永远无法再被复制。无论你的计算机系统在运行什么（这是你无法控制的），原则上都应该可以忽略。

难分伯仲。通常，当我们比较同一个任务的两个不同程序时，一个可能在某些情况下更快，而在其他情况下更慢。上述讨论的一个或多个因素可能会对此有影响。另外，一些程序员（以及某些学生）存在一种自然的倾向，就是喜欢投入大量精力运行这种赛马程序来找到“最佳”的实现，但这样的工作最好留给专家去做。

强烈依赖输入值。我们确定一个程序运行时间的增长量级的假设之一是运行时间应该主要取决于问题的规模（而与输入值无关）。如果情况并非如此，则可能会得到不一致的结果，或无法验证我们的假设。我们采用的例子 ThreeSum 不存在这个问题，但是我们编写的许多程序确实会产生这个问题。读者可以在本章中看到几个这样的程序。通常，设计的主要目标

是消除这种对输入值的依赖。如果不能实现这个目标，则必须为输入数据建立一个细致的模型，用于指导程序行为的分析，这可能是一个更大的挑战。例如，如果我们正在编写一个处理基因组的程序，如何判断其针对不同的基因组的效果？一个良好地描述自然界基因组的模型正是科学家所寻求的，所以预测我们的程序在自然界中发现的数据上的运行时间将对模型做出有价值的贡献！

510

多个问题参数。我们一直关注使用单一参数 n 的函数来测量性能，这个参数通常是命令行参数的值或输入的规模。但是存在多个这样的参数来确定待处理的数据量的情况并不罕见。例如，假设 $a[]$ 是长度为 m 的数组，而 $b[]$ 是长度为 n 的数组。考虑以下代码片段，它计算 $a[i]+b[j]$ 等于 0 的所有（无序） i 和 j 数对的数量：

```
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        if (a[i] + b[j] == 0)
            count++;
```

运行时间的增长量级取决于两个参数—— m 和 n 。在这种情况下，我们分别处理参数，在研究一个参数的时候将另一个参数固定。例如，上述代码片段运行时间的增长量级是 $m \times n$ 。同样，LongestCommonSubsequence（程序 2.3.6）包含两个参数 m （第一个字符串的长度）和 n （第二个字符串的长度），因此，其运行时间的增长量级是 $m \times n$ 。

尽管有这些注意事项，但理解每个程序运行时间的增长量级对于每个程序员而言都是有价值的知识，而且我们刚才所阐述的方法非常强大并且广泛适用。Knuth 的观点是，原则上我们可以通过使用这些方法来实现详细、准确的预测。典型的计算机系统非常复杂，精密的分析最好交给专家，但同样的方法对于近似估计任何程序的运行时间都是有效的。就像火箭科学家需要了解某次测试发射会降落在海洋还是城市，医学研究者需要知道药物试验会杀死还是治愈所有的受试者，使用计算机程序的科学家和工程师则需要知道程序会运行一秒还是一年。

511

性能保证 对于某些程序，我们要求对于给定规模的任何输入，程序的运行时间都小于某个界限。为了提供这样的性能保证，理论家希望得到一种最悲观的结论：在最坏的情况下，程序运行时间会是多少？

例如，这种保守的方法可能适用于运行核反应堆、空中交通管制系统或汽车刹车软件。我们必须保证这类软件能在我们设定的时间内完成工作，否则将会导致灾难性后果。科学家在研究自然界时通常不会考虑最坏的情况：在生物学中，最坏的情况也许是人类的灭绝；在物理学中，最坏的情况可能是宇宙的终结。但是在计算机系统中，最糟糕的情况是我们真正需要关心的事情，因为输入是由另一个（潜在恶意）用户生成的，而不是自然产生的。例如，没有使用性能保证算法的网站常遭受拒绝服务（denial-of-service）攻击，黑客通过发送大量洪水般的请求，使网站造成灾难性的性能下降。

性能保证很难用科学的方法来验证，因为我们无法使用所有可能的输入测试一个假设，比如对于增长量级为线性对数型的归并排序算法，可能的输入组合太多，不可能一一尝试。我们可能会提供一组输入让归并排序很慢，但是如何能证明这就是最慢的情况呢？我们必须使用数学分析，而不能使用实验方法。

算法分析人员的任务是尽可能多地发现有关算法的信息，应用程序员的任务是应用这些知识来开发有效解决问题的程序。例如，如果你正在使用二次时间算法来解决问题，但是后

来找到能够保证最差情况为线性对数时间的算法，显然你会更喜欢线性对数时间算法。在极少数情况下，你可能更喜欢二次时间算法，因为对于你需要解决的某种输入，二次型算法运行更快，或者因为线性对数算法的实现太复杂。

理想的情况下，我们需要能够让代码显得清晰紧凑的算法，它既提供了良好的最差情况保证，又在常见的输入上有良好的性能。我们在本章讨论的许多经典算法对于多种应用都具有重要意义，因为它们具有上述特性。使用这些算法作为模型，读者可以针对编程时遇到的

[512] 典型问题，自行设计好的解决方案。

内存 与运行时间一样，程序的内存使用与物理世界直接关联：因为内存存在计算机中占据了相当大量的电路，是实现程序的数据存储和随后访问的关键物理介质。在给定时间内，你需要存储的值越多，所需的电路就越多。为了控制成本，你必须关注内存的使用情况。你也许已经意识到计算机内存使用的一些限制（有时甚至比时间限制更重要），因为你可能需要花费额外的金钱来获取更多的内存。

在计算机上 Java 很好地定义了内存使用（每次运行程序时，每个值占用同样大小的内存），但是 Java 可以在各种计算设备上实现，并且内存消耗与实现有关。为了简单起见，我们使用术语“典型的”来表示受机器影响的值。在一个典型的 64 位机器上，计算机内存被划分成字，每 64 位字由 8 字节组成，每字节包含 8 位，每位是一个二进制数字。

分析内存使用情况与分析时间使用情况有些不同，主要是因为 Java 最重要的功能特性之一就是内存分配系统，其设计目标是减轻管理内存的繁重工作。目前，建议读者在适当的时候充分利用这个功能。尽管如此，你仍有责任了解，或者至少大概了解，程序的内存需求何时会妨碍用户解决给定问题。

基本类型。估算简单程序的内存使用情况很容易，就像我们在第 1 章中讨论的那样：计算变量的数目，并根据变量的类型乘以各变量占用的字节数。例如，由于 Java 的 int 数据类型代表了在 -2 147 483 648 和 2 147 483 647 之间的整数值集合，总共有 2^{32} 个不同的取值，典型的 Java 实现采用 32 位（4 字节）来表示每个 int 值。类似地，典型的 Java 实现用 2 字节（16 位）表示 char 值，用 8 字节（64 位）表示 double 值，用 1 字节表示 boolean 值（因为计算机一次最少就是访问 1 字节）。例如，如果你的计算机有 1GB 的内存（大约 10 亿字节），那么任何时候，内存都无法容纳多于 2.56 亿个 int 值或者 1.28 亿个 double 值。

类型	字节数
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

一些基本类型的典型内存需求

[513] **对象。**为了确定对象的内存使用情况，我们要计算每个实例变量使用的内存大小，同时还要加上每个对象产生的额外开销，通常为 16 字节。内存通常被填充（向上取整）为 8 字节的倍数——当前硬件中内存“字”的整数倍。

例如，在一个典型的系统中，一个 Complex（程序 3.2.6）对象使用 32 字节（16 字节的开销，以及两个 double 型变量，每个占据 8 字节）。由于许多程序创建了数百万个 Color 对象，因此一种典型的 Java 实现是将其所需的信息打包为一个 32 位的 int 值。所以，一个 Color 对象使用 24 字节（16 字节的开销，4 字节用于存储 int 实例变量，4 字节用于填充）。

对象引用通常使用 8 字节（1 个字）的内存。当一个类使用一个对象引用作为实例变量时，必须将对象引用的内存（8 字节）和对象本身所需的内存分开计算。例如，Body（程序 3.4.1）对象使用 168 字节：对象本身的开销（16 字节）、存储一个 double 值的开销（8 字节）

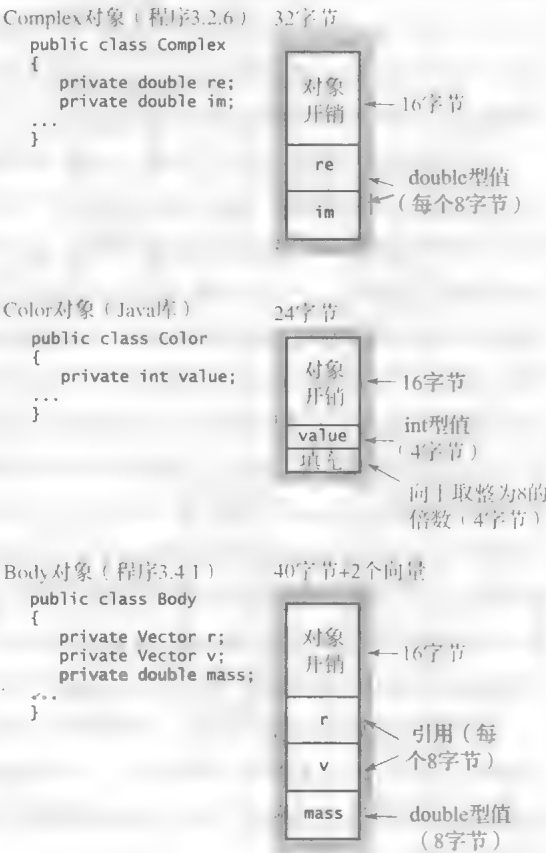
和两个引用的开销（每个 8 字节），再加上 Vector 对象所需的内存（Vector 对象的内存消耗我们稍后讨论）。

数组。Java 中的数组作为对象实现，通常使用一个 int 实例变量表示数组长度。对于基本数据类型，一个由 n 个元素组成的数组需要 24 字节的内存开销（16 字节的对象开销、4 字节的长度、4 字节的填充），再加上 n 倍存储每个元素所需的字节数。例如，Sample（程序 1.4.1）中的 int 数组使用 $4n+24$ 字节的开销；Coupon（程序 1.4.2）中的布尔数组使用 $n+24$ 字节的开销。请注意，一个布尔数组的每个元素消耗 1 字节的内存（浪费了 8 位中的 7 位）——通过一些额外的手段，我们可以让每个元素只使用 1 位来完成这个工作（参见练习 4.1.26）。

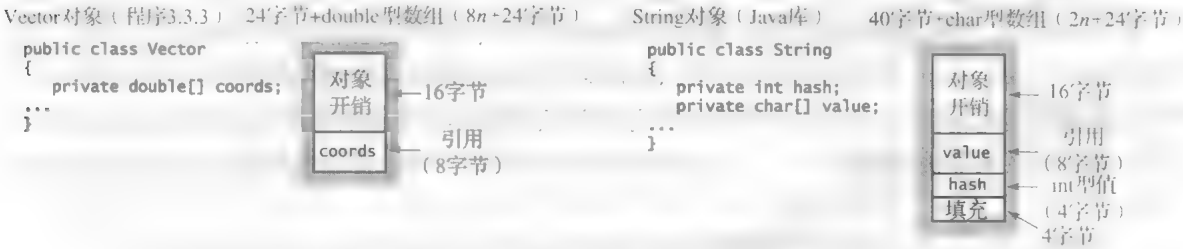
一个对象数组是对象引用的数组，所以我们需要同时考虑引用的内存和对象的内存。例如，一个包含 n 个 Charge 对象的数组消耗 $48n+24$ 字节：数组开销（24 字节）、Charge 的引用（8 字节）和 Charge 对象的内存（40 字节）。这个分析假设所有的对象都是不同的：多个数组元素可能引用相同的 Charge 对象（别名）。

类 Vector（程序 3.3.3）包含一个数组作为实例变量。在典型的系统中，长度为 n 的 Vector 对象需要 $8n+48$ 字节的开销：对象开销（16 字节）、对 double 数组的引用（8 字节）和 double 数组所占的内存（ $8n+24$ 字节）。因此，Body 中的每个 Vector 对象都使用 64 字节的内存（因为 $n=2$ ）。

字符串对象。我们使用与其他对象相同的方式来计算 String 对象消耗的内存。长度为 n 的 String 对象通常消耗 $2n+56$ 字节：对象开销（16 字节）、对 char 数组的引用（8 字节）、char 数组所占的内存（ $2n+24$ 字节）和一个 int 型变量（1 字节）及填充的开销（4 字节）。String 对象中的 int 实例变量是一个散列码，其在某些情况下可以避免重复计算，现在还不关心它的用途。如果字符串中的字符数不是 4 的倍数，则会填充字符数组的内存，以使 char 数组的字节数为 8 的倍数。



对象的典型内存需求



Vector 和 String 对象的典型内存需求

二维数组。正如我们在 1.4 节看到的, Java 中的二维数组是数组的数组。因此, Markov (程序 1.6.3) 中的二维数组消耗 $8n^2+32n+24$ (或 $\sim 8n^2$) 字节: 数组中每个数组的开销 (24 字节)、对行数组的 n 个引用 ($8n$ 字节) 和 n 行数组 (每个 $8n+24$ 字节)。如果数组元素是对象, 即数组由用对象的引用填充的数组构成, 通过近似的计算得到 $\sim 8n^2$ 字节, 这里我们需要把这些对象本身占据的内存也算在内。

这些基本的机制适用于估算大多数程序的内存使用情况。但是存在很多复杂的因素会导致内存估算十分困难。我们已经注意到了缓存可能带来的潜在影响。此外, 当涉及函数调用时, 内存消耗是一个复杂的动态过程, 系统内存分配机制扮演着更重要的角色, 并且具有系统依赖性。例如, 当程序调用一个函数时, 系统从一个

称为栈的特殊内存区分配函数 (对于函数的局部变量) 所需的内存, 当函数返回给调用者时, 内存被回收到栈。由于这个原因, 在递归函数中创建数组或其他大对象是一件很危险的事, 因为每个递归调用都意味着大量的内存消耗。当你用 `new` 创建一个对象时, 系统从一个称为堆的特殊区域为对象分配所需的内存, 必须记住每个对象都会一直存在, 直到没有引用指向该对象, 此时一个称为垃圾回收的系统进程会从堆中回收内存。这种动态过程会使得精确估计程序的内存占用量变得十分困难。

类型	字节数
<code>boolean[]</code>	$n + 24 \sim n$
<code>int[]</code>	$4n + 24 \sim 4n$
<code>double[]</code>	$8n + 24 \sim 8n$
<code>Charge[]</code>	$40n + 24 \sim 40n$
<code>Vector</code>	$8n + 48 \sim 8n$
<code>String</code>	$2n + 56 \sim 2n$
<code>boolean[][]</code>	$n^2 + 32n + 24 \sim n^2$
<code>int[][]</code>	$4n^2 + 32n + 24 \sim 4n^2$
<code>double[][]</code>	$8n^2 + 32n + 24 \sim 8n^2$

516

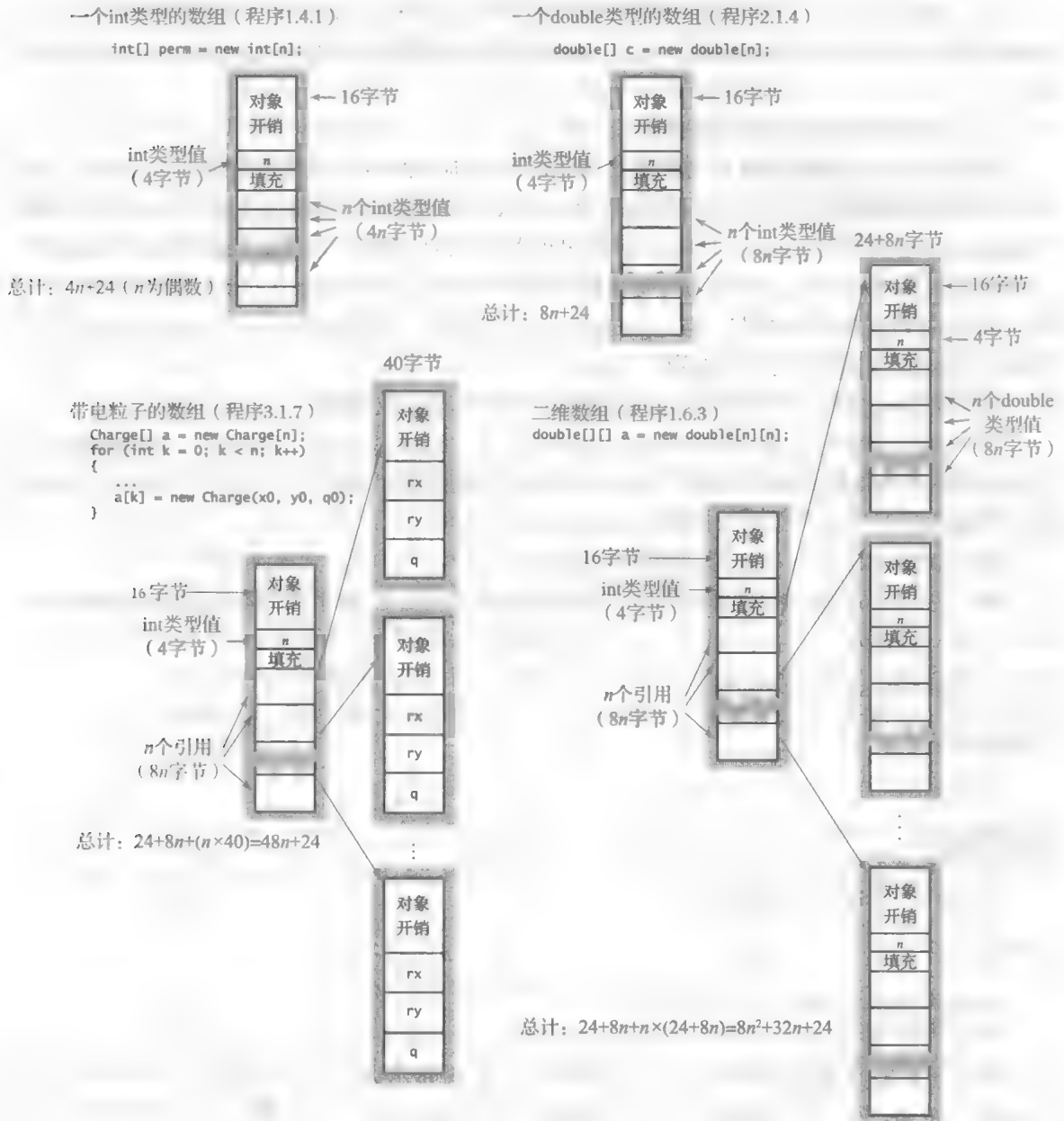
可变长度数据类型的典型内存需求

展望 良好的性能对于一个程序十分重要。运行起来慢得不可思议的程序几乎等同于一个错误的程序, 都没有什么用处, 所以从一开始就要注意成本, 以便我们能够了解可以解决哪些类型的问题。特别地, 理解构成程序内循环的代码总是明智的。

在程序设计中, 也许最常犯的错误就是过度关注性能。程序设计的首要任务是使代码清晰且正确。如果修改程序的唯一目的是加速程序, 那么这件事最好留给专家。事实上, 这样做的结果往往适得其反, 因为这种情况下创建的代码往往复杂并且难以理解。C. A. R. Hoare (快速排序算法的发明者, 也是编写清晰且正确的代码的主要支持者) 曾经总结了 this 观点, 他说: “不成熟的优化是一切错误的根源。” Knuth 为这句话加上了限定词 “在 (至少大多数) 程序设计中”。除此之外, 如果可以节省的成本有限, 则改善运行时间是不值得的。例如, 如果运行时间只是一个瞬间, 则将程序的运行速度提高 10 倍是没有什么意义的。即使一个程序的运行时间需要几分钟, 实现和调试一个改进算法所需的总时间也可能远远超过仅仅运行一个稍微慢的程序所需要的时间——此时最好让计算机承担这项工作。更糟糕的是, 你可能花费大量的时间和精力思考改进程序, 但实际上并没有任何效果。

在开发一个算法时第二个常犯的错误是忽略性能特征。更快的算法通常比粗糙的解决方案复杂得多, 所以你可能趋向于接受一个较慢的算法来避免处理更复杂的代码。但是, 有时通过几行优秀的代码就可以节省巨大的资源。即使线性或线性对数算法仅仅稍微复杂一些, 也能用更短时间解决问题, 但是大部分计算机系统用户却选择花费大量时间来等待简单的二次型算法来完成问题的求解。只有处理大型的问题时我们才别无选择, 只能寻求更好的算法。

改善程序使其更加清晰、高效和优雅应该作为我们每次程序设计的目标。如果开发一个程序的整个过程中我们一直关注成本, 那么每次使用这个程序时都会从中受益。



int 值、double 值、对象和数组的典型内存需求

517
518

问答环节

问：我如何知道在我的计算机上两个浮点数的加法或者乘法需要消耗多长时间？

答：运行一些实验，你自己就能找到答案！本书官网上的 TimePrimitives 程序使用 Stopwatch 来测试不同算术运算在基本类型上的执行时间。这种技术也可用于测量程序实际运行的时间，就像观测挂钟一样。如果你的系统没有运行许多其他程序，这可以产生比较准确的结果。读者可以在本书官网上找到更多关于改进这些实验的信息。

问：调用 Math.sin()、Math.log() 和 Math.sqrt() 需要多少时间？

答：运行一些实验，你自己就能找到答案！利用 Stopwatch 可以很容易编写类似于 TimePrimitives 的程序来回答这类问题，如果习惯了这些操作，你将能够更加有效地使用计算机。

问：字符串操作需要多长时间？

答：运行一些实验，你自己就能找到答案！（你应该能接收到我们想传达的信息了吧？）String 数据类型的实现允许方法 length() 和 charAt() 在常数时间内运行。诸如 toLowerCase() 和 replace() 之类方法的运行时间与字符串的长度线性相关。方法 compareTo()、equals()、startsWith() 和 endsWith() 需要的时间与解析答案所需的字符数成正比（最好的情况是常数时间，最差的情况是线性时间），但方法 indexOf() 会很慢。字符串拼接以及 substring() 方法需要的时间正比于结果中的字符总数。

问：为什么分配一个大小为 n 的数组消耗的时间与 n 成正比？

答：Java 把数组元素自动初始化为默认值（0、false 或 null）。原则上，如果计算机将每个元素的初始化推迟到程序第一次访问该元素，这会是个常数时间的操作。但是大多数的 Java 实现通过遍历整个数组来初始化每个元素。

519

问：如何确定 Java 程序可用的内存容量？

答：内存不足时 Java 会告诉你，可以通过运行一些实验来验证。例如，使用 PrimeSieve（程序 1.4.3），输入

```
% java PrimeSieve 100000000
```

并得到结果

```
50847534
```

接着又输入

```
% java PrimeSieve 1000000000
```

并得到结果

```
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
```

结果表明有足够的空间来存放一个大小为 1 亿的布尔数组，但无法存放一个大小为 10 亿的布尔数组。你可以使用命令行选项增加分配给 Java 的内存容量。以下命令使用命令行参数 1 000 000 000 和命令行选项 “-Xmx1110mb” 来执行 PrimeSieve，该命令要求最多 1 100MB 内存（如果系统中有足够的可用内存）。

```
% java -Xmx1100mb PrimeSieve 1000000000
```

问：当有人说最坏运行时间为 $O(n^2)$ 时，意味着什么？

答：这是一种被称为大 O 的表示法。如果存在常量 c 和 n_0 ，对于所有的 $n > n_0$ 满足 $|f(n)| \leq c|g(n)|$ ，则可将 $f(n)$ 表示为 $O(g(n))$ 。换句话说，函数 $f(n)$ 的上限是 $g(n)$ ，这时我们忽略了常量因子，并假设 n 的值足够大。例如，函数 $30n^2 + 10n + 7$ 是 $O(n^2)$ 。对于所有可能的输入，如果一个输入大小为 n 的函数运行时间为 $O(g(n))$ ，那么算法的最差运行时间为 $O(g(n))$ 。大 O 表示法和最差运行时间被理论计算机科学家广泛用于证明算法的理论依据，所以如果你选修了算法和数据结构课程，肯定见过这个符号。

520

问：我们是否可以使用算法的最差情况运行时间 $O(n^3)$ 或者 $O(n^2)$ 来预测性能？

答：不可以，因为实际运行时间可能会少得多。例如，函数 $30n^2+10n+7$ 是 $O(n^2)$ ，但也是 $O(n^3)$ 和 $O(n^{10})$ ，因为大 O 表示法只提供最差情况运行时间的上限。而且，即使存在一些输入，其运行时间正比于给定函数，但也许这些输入在实际应用中也不会遇到。因此，不能使用大 O 表示法来预测性能。我们使用的波浪线表示法和增长量级分类法比大 O 表示法更精确，因为它们提供了与函数增长量级匹配的上限和下限。许多程序员使用大 O 表示法指示相对应的上限和下限是不正确的。

521

练习

- 4.1.1 为 ThreeSum（程序 4.1.1）实现静态方法 printTriples()，将所有和为 0 的三元组打印到标准输出。
- 4.1.2 修改 ThreeSum，使其接收一个整数命令行参数 target，并在标准输入上找出总和最接近 target 的三元组。
- 4.1.3 编写一个从标准输入中读取 long 类型的程序 FourSum，并统计总和为 0 的四元组的个数。使用四重嵌套循环。请问程序运行时间的增长量级是多少？请估计程序在一个小时内可以处理的最大的输入规模 n 为多少。然后，运行程序来验证你的假设。
- 4.1.4 请通过归纳法证明 0 到 $n-1$ 之间的所有（无序）整数对的数量为 $n(n-1)/2$ ，然后使用归纳法证明 0 到 $n-1$ 之间的所有（无序）整数三元组数量为 $n(n-1)(n-2)/6$ 。

二元组答案：这个公式对于 $n=1$ 是正确的，因为有 0 对。对 $n>1$ ，计算不包括 $n-1$ 的所有数对，根据归纳法假设个数是 $(n-1)(n-2)/2$ ，计算含有 $n-1$ 的所有数对，就是 $n-1$ 个，因此得到二元组个数为

$$(n-1)(n-2)/2+(n-1)=n(n-1)/2$$

三元组答案：对于 $n=2$ ，公式成立。对于 $n>2$ ，计算不包括 $n-1$ 的所有三元组，根据归纳法假设个数是 $(n-1)(n-2)(n-3)/6$ ，计算含有 $n-1$ 的所有数对，就是 $(n-1)(n-2)/2$ 个，因此得到三元组个数为

$$(n-1)(n-2)(n-3)/6+(n-1)(n-2)/2=n(n-1)(n-2)/6$$

- 4.1.5 用积分近似证明 0 和 n 之间所有不同的整数三元组数目为 $n^3/6$ 。
- 答案： $\sum_0^n \sum_0^i \sum_0^j 1 \approx \int_0^n \int_0^i \int_0^j dk \, dj \, di = \int_0^n \int_0^i j \, dj \, di = \int_0^n (i^2/2) \, di = n^3/6$
- 4.1.6 证明函数 cn^b 的 log-log 图斜率为 b ，并且 x 轴截距为 $\log c$ ，那么 $4n^3(\log n)^2$ 的斜率和 x 轴截距是多少？
- 4.1.7 运行以下代码片段，count 的值（作为 n 的一个函数）是多少？

522

```
long count = 0;
for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
        for (int k = j + 1; k < n; k++)
            count++;
```

答案： $n(n-1)(n-2)/6$

- 4.1.8 使用波浪线表示法简化以下公式，并且给出其增长量级：
- a. $n(n-1)(n-2)(n-3)/24$ b. $(n-2)(\lg n-2)(\lg n+2)$ c. $n(n+1)-n^2$
- d. $n(n+1)/2+n \lg n$ e. $\ln((n-1)(n-2)(n-3))^2$
- 4.1.9 根据标准输入的整数 n 确定 ThreeSum 中以下语句运行时间的增长量级：

```
int[] a = StdIn.readAllInts();
```

答案：线性。运行时间的瓶颈是隐式数组初始化和隐藏在 `readAllInts` 中的输入读取循环。这样的输入循环的成本可以控制在线性时间内完成，但是如果处理的数据量非常庞大，甚至可能会需要平方级的时间成本。

4.1.10 确定下面的代码段是线性型、二次型还是三次型（作为 n 的函数）的增长量级。

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        if (i == j) c[i][j] = 1.0;
        else      c[i][j] = 0.0;
```

523

4.1.11 假设一个算法基于输入大小为 1000、2000、3000 和 4000 的运行时间分别是 5 秒、20 秒、45 秒和 80 秒。请估计需要多长时间才能解决问题规模为 5000 的问题。请问算法是线性、线性对数、二次、三次还是指数型的增长量级？

4.1.12 你更喜欢哪种算法，增长量级是二次、线性对数还是线性型？

答案：虽然很容易根据增长量级做出快速选择，但这样选择很容易被误导。你需要了解问题的规模以及运行时间的首项系数相对值的大小。例如，假设运行时间为 n^2 秒、 $100n \log_2 n$ 秒和 $10\,000n$ 秒。 n 小于或等于 1000 时，二次型算法最快；线性算法永远不会比线性对数算法更快（ n 必须大于 2^{100} 时线性算法会更快，但如此巨大的数一般不予考虑）。

4.1.13 应用科学的方法分析下面的代码片段（作为参数 n 的函数）的运行时间的增长量级的假设，并验证你的假设：

```
public static int f(int n)
{
    if (n == 0) return 1;
    return f(n-1) + f(n-1);
}
```

4.1.14 应用科学的方法开发和验证一个 Coupon（程序 2.1.3）中 `collect()` 方法（作为参数 n 的函数）的运行时间的增长量级的假设。注意：倍增方法对于区分线性型和线性对数型无效，你可以尝试将输入大小进行平方。

4.1.15 应用科学的方法开发和验证 Markov（程序 1.6.3）运行时间的增长量级的假设，它应该是实验次数 `trials` 和 n 的函数，而 `trials` 和 n 是从命令行参数获得的。

524

4.1.16 应用科学的方法开发和验证下面两个代码片段的运行时间的增长量级的假设，它们应该是参数 n 的函数。

```
String s = "";
for (int i = 0; i < n; i++)
    if (StdRandom.bernoulli(0.5)) s += "0";
    else                          s += "1";

StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++)
    if (StdRandom.bernoulli(0.5)) sb.append("0");
    else                          sb.append("1");
String s = sb.toString();
```

4.1.17 如下四个 Java 函数均返回一个长度为 n 、其字符均为 x 的字符串。请确定每个函数的运行时间的增长量级。回顾一下，在 Java 中拼接两个字符串需要的时间与结果字符串的长度成正比。

```
public static String method1(int n)
{
    if (n == 0) return "";
```

```

    String temp = method1(n / 2);
    if (n % 2 == 0) return temp + temp;
    else return temp + temp + "x";
}

public static String method2(int n)
{
    String s = "";
    for (int i = 0; i < n; i++)
        s = s + "x";
    return s;
}

public static String method3(int n)
{
    if (n == 0) return "";
    if (n == 1) return "x";
    return method3(n/2) + method3(n - n/2);
}

public static String method4(int n)
{
    char[] temp = new char[n];
    for (int i = 0; i < n; i++)
        temp[i] = 'x';
    return new String(temp);
}

```

525

- 4.1.18 下面的代码片段（改编自《Java 程序设计》一书）创建了从 0 到 $n-1$ 之间整数的一个随机排列。请确定其运行时间的增长量级，它应该是 n 的函数。将其增长量级与 1.4 节中混排代码的增长量级进行比较。

```

int[] a = new int[n];
boolean[] taken = new boolean[n];
int count = 0;
while (count < n)
{
    int r = StdRandom.uniform(n);
    if (!taken[r])
    {
        a[r] = count;
        taken[r] = true;
        count++;
    }
}

```

526

- 4.1.19 以下两个函数的运行时间的增长量级是什么？每个函数都将一个字符串作为参数，并返回该字符串的逆序。

```

public static String reverse1(String s)
{
    int n = s.length();
    String reverse = "";
    for (int i = 0; i < n; i++)
        reverse = s.charAt(i) + reverse;
    return reverse;
}

public static String reverse2(String s)
{
    int n = s.length();
    if (n <= 1) return s;
}

```

```
String left = s.substring(0, n/2);
String right = s.substring(n/2, n);
return reverse2(right) + reverse2(left);
}
```

4.1.20 给出一个线性时间算法来输出一个字符串的逆序。

答案:

```
public static String reverse(String s)
{
    int n = s.length();
    char[] a = new char[n];
    for (int i = 0; i < n; i++)
        a[i] = s.charAt(n-i-1);
    return new String(a);
}
```

4.1.21 请编写一个程序 `MooresLaw`，带一个命令行参数 n ，如果处理器的速度每 n 个月就会翻一倍，请输出十年后处理器的速度增长了多少。如果速度每 n ($=15$) 个月增加一倍，则十年后处理器的速度增加了多少？如果 $n=24$ 个月呢？

527

4.1.22 请使用正文中的 64 位内存模型，给出第 3 章中以下数据类型的每个对象的内存需求：

- a. `Stopwatch` b. `Turtle` c. `Vector`
- d. `Body` e. `Universe`

4.1.23 请估计垂直渗透检测（程序 2.4.2）的 `PercolationVisualizer`（程序 2.4.3）使用的空间总量，它应该是网格大小 n 的函数。加分题：对递归渗透检测方法（程序 2.4.5）回答同样的问题。

4.1.24 请估计你的计算机可以容纳的最大二维整数数组的大小，然后尝试分配这样一个数组。

4.1.25 请估计 `CompareDocuments`（程序 3.3.5）的内存使用量，它应该是文档数量 n 和维度 d 的函数。

4.1.26 请编写 `PrimeSieve`（程序 1.4.3）的一个版本，其使用一个整数数组而不是布尔数组，并使用每字节中的所有位，从而增大 n 的最大值，使得程序能够处理的问题可以增大 8 倍。

4.1.27 下表列举了在 n 值不同时三个程序的运行时间。根据所提供的信息，填入你的合理估计值。

程序	1000	10 000	100 000	1 000 000
A	0.001 s	0.012 s	0.16 s	?
B	1 min	10 min	1.7 h	?
C	1 s	1.7 min	2.8 h	?

528

请给出每个程序运行时间的增长量级假设。

创新练习

4.1.28 三数和分析。计算 n 个随机 32 位整数中不存在总和为 0 的三元组的概率。加分题：给出三元组的期望个数的近似公式（作为 n 的函数），并运行实验来验证你的假设。

4.1.29 最近数对。设计一个二次型算法，给定一个整数数组，找出数组中值最接近的一对数字。（在下一节中，你将被要求为这个问题实现一个线性对数算法。）

4.1.30 beck 漏洞攻击。假设有一个流行的 Web 服务器支持一个函数 `no2slash()`，其目的是去除重复的字符“/”。例如，字符串 `/d1///d2////d3/test.html` 会转换为 `/d1/d2/d3/test.html`。原始算法是反复查找字符“/”并复制字符串的其余部分：


```

int n = name.length();
int i = 1;
while (i < n)
{
    if ((c[i-1] == '/') && (c[i] == '/'))
    {
        for(int j = i+1; j < n; j++)
            c[j-1] = c[j];
        n--;
    }
    else i++;
}

```

遗憾的是，上述代码可能需要二次级时间（例如，如果字符串含有重复 n 次的“/”字符）。通过同时发送大量包含“/”字符的请求，黑客可能淹没服务器，使其他进程无法获得足够的 CPU 时间，从而造成拒绝服务攻击。开发一个线性时间运行的 `no2slash()` 版本，从而有效防御这种类型的攻击。

4.1.31 子集和。请编写一个从标准输入中读取长整型的程序 `SubsetSum`，用于统计那些总和恰好为 0 的整数子集数目。给出程序运行时间的增长量级。

529

4.1.32 杨氏矩阵。假设存在一个 $n \times n$ 的二维整数数组 $a[i][j]$ ，对于所有的 i 和 j ，满足 $a[i][j] < a[i+1][j]$ 和 $a[i][j] < a[i][j+1]$ ， 5×5 数组如下所示。

```

5  23  54  67  89
6  69  73  74  90
10 71  83  84  91
60 73  84  86  92
99 91  92  93  94

```

具有这个属性的二维数组被称为杨氏矩阵。请编写一个函数，该函数将一个 $n \times n$ 的杨氏矩阵和一个整数作为参数，并确定该整数是否在杨氏矩阵中。函数运行时间的增长量级应该与 n 呈线性关系。

4.1.33 数组旋转。给定一个包含 n 个元素的数组，请给出一个线性时间算法，用于旋转数组的 k 个位置。也就是说，如果数组包含 n 个元素 a_0, a_1, \dots, a_{n-1} ，则旋转后的数组为 $a_k, a_{k+1}, \dots, a_{n-1}, a_0, \dots, a_{k-1}$ 。最多使用一个常量大小的额外空间。提示：反转三个子数组。

4.1.34 查找重复的整数。(a) 给定包含 n 个整数（1 到 n ）的数组，其中一个整数重复两次，一个整数缺失，请给出一个算法，使用线性时间和常量额外内存找到缺失的整数。不考虑整数溢出。(b) 给定一个包含 n 个整数的只读数组，其中 1 到 $n-1$ 的每个值只出现一次，有一个整数出现了两次，请给出一个算法，使用线性时间和常量额外内存找到重复的整数。(c) 给定一个包含 n 个整数（1 到 $n-1$ ）的只读数组，请给出一个算法，使用线性时间和常量额外内存查找重复值。

4.1.35 阶乘。请设计一个快速算法用于计算 n 为很大值时的 $n!$ ，使用 Java 的 `BigInteger` 类。使用你的程序计算 $1\,000\,000!$ 中最多有多少个连续的 9。为你的算法运行时间的增长量级提出一个假设并验证。

530

4.1.36 最大和。设计一个线性时间算法，用于在包含 n 个长整型的数组中找到最大 m 个数的连续子数组，使得这个子数组在所有这样的子数组中相加和最大。请实现你的算法，并确保程序运行时间的增长量级是线性的。

4.1.37 最大平均值。请编写一个程序，在包含 n 个长整型的数组中查找最多 m 个元素的连续子数组，使其平均值在所有这样的子数组中最大。使用科学的方法证明程序运行时间的增长量级为

mn^2 。接下来,请编写一个程序解决如下问题,首先对于每个 i 计算 $\text{prefix}[i]=a[0]+\dots+a[i]$,然后使用表达式 $(\text{prefix}[j]-\text{prefix}[i])/(j-i+1)$ 计算区间 $a[i]$ 到 $a[j]$ 的平均值。请使用科学的方法来证明这种方法把增长量级减少了 n 倍。

- 4.1.38 模式匹配。给定由黑色(1)和白色(0)像素组成的 $n \times n$ 数组,请设计一个线性时间的算法,查找不包含白色像素的最大方形子矩阵。在下面的例子中,最大的黑色像素子矩阵是用浅色显示的 3×3 的子矩阵。

```
10111000
00010100
00111000
00111010
00111111
01011110
01011010
00011110
```

请实现你的算法,并确保其运行时间的增长量级与像素数量呈线性关系。加分题:请设计一个算法来找到最大矩形黑色子矩阵。

- 4.1.39 次指数函数。请查找一个函数,其运行时间的增长量级比多项式函数大,但比指数函数小。
加分题:请编写一个程序,其运行时间是这种增长量级的。

531

4.2 排序和搜索

排序问题旨在将数组中的元素重新排列成升序。在许多计算和应用中,这是一个常用且关键的任务:音乐库中的歌曲按字母顺序排列;电子邮件按接收时间的相反顺序显示等。把事物按某种顺序排列是一种很自然的需求。排序非常有用的一个理由是,在排序的列表中搜索某些内容,要比在未排序的列表中容易得多。这种需求在计算中尤为突出,因为计算中要搜索的数组可能十分巨大,而高效的搜索可能是一个问题解决方案中的关键因素。

排序和查找对于商业应用程序(企业按照顺序存储客户文件)和科学应用程序(组织数据和计算)非常重要,并且在其他可能看似与事物排序无关的领域中也有各种应用,包括数据压缩、计算机图形学、生物信息学、数值计算、组合优化、密码学等。

我们使用这些基本问题来阐述一个观念,那就是高效的算法是计算问题的有效解决方案的关键之一。事实上,科学家已经提出了许多不同的排序和查找方法。解决一个具体任务时,我们应该选择哪种算法呢?这是一个非常重要的问题,因为不同的算法在性能上有显著的差异,从而导致成功解决实际问题还是根本无法解决实际问题的差别(即使使用最快的计算机)。

在本节中,我们将详细讨论用于查找和排序的两个经典算法——二分查找和归并排序,以及若干侧重效率的应用程序。通过这些例子,你会意识到,在解决需要大量计算的问题时,不仅仅需要注重方法的有效性,还需要注意运行成本。

532

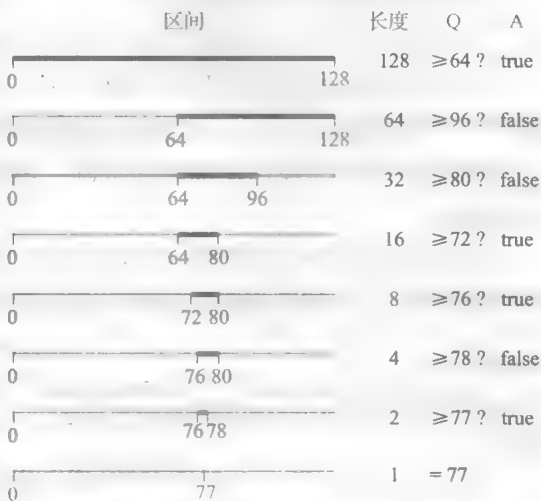
二分查找法 “20 个问题”的游戏(见程序 1.5.2)为计算问题设计高效算法提供了一个重要并且有用的思路。规则十分很简单:你的任务是猜测一个神秘整数的值,它在 0 到 $n-1$ 这 n 个整数之中。每次你做出一个猜测时,程序会告知猜测的值等于、大于或者小于那个神秘数字。为了阐述得更加清楚,我们首先稍微修改一下游戏的提问形式:“请问该数大于或等于 x 吗?”,其答案仅为 true 或 false,假设 n 是 2 的整数次幂。

正如在 1.5 节中所述,对于这个问题最有效的策略是不断维护和修正包含秘密数字的区

间。在每个步骤中，都可以通过提问将区间的大小缩小一半。具体来说，我们猜测在数值区间中间的那个数字，根据猜测结果把包含秘密数字的区间减半。更准确地说，我们使用一个半开区间，半开区间包含左端点但不包含右端点。我们使用符号 $[lo, hi)$ 来表示所有大于等于 lo 但小于（不等于） hi 的整数。开始时 $lo=0$ 、 $hi=n$ ，使用如下递归策略：

- 基础步骤：如果 $hi-lo=1$ ，那么秘密数字是 lo 。
- 归纳步骤：否则，询问秘密数字是否大于或等于中间数 $mid=lo+(hi-lo)/2$ 。如果是，则继续在区间 $[mid, hi)$ 中查找；否则在 $[lo, mid)$ 中查找。

程序 Questions（程序 4.2.1）中的 `binarySearch()` 函数是该策略的一个实现。这种策略是二分查找的通用问题求解算法的例子，二分查找法具有广泛的应用。



用二分查找法找到一个秘密的数字

533

程序4.2.1 二分查找法（20个问题游戏）

```
public class Questions
{
    public static int binarySearch(int lo, int hi)
    {
        // 在[lo, hi) 区间中找数字
        if (hi - lo == 1) return lo;
        int mid = lo + (hi - lo) / 2;
        StdOut.print("Greater than or equal to " + mid + "? ");
        if (StdIn.readBoolean())
            return binarySearch(lo, mid);
        else
            return binarySearch(mid, hi);
    }

    public static void main(String[] args)
    {
        // 开始20个问题的游戏
        int k = Integer.parseInt(args[0]);
        int n = (int) Math.pow(2, k);
        StdOut.print("Think of a number ");
        StdOut.println("between 0 and " + (n-1));
        int guess = binarySearch(0, n);
        StdOut.println("Your number is " + guess);
    }
}
```

lo	可能的最小值
hi - 1	可能的最大值
mid	中点
k	提问的数量
n	可选值的数量

这段代码使用二分查找法实现与程序1.5.2相同功能的的游戏程序，但角色相反：由用户选择一个秘密数字，程序猜测其值。程序带一个整数型命令行参数 k ，提示用户考虑一个0到 $n-1$ 之间的数字，其中 $n=2^k$ ，然后用 k 次提问猜出正确答案。

```
% java Questions 7
Think of a number between 0 and 127
Greater than or equal to 64? false
Greater than or equal to 96? true
Greater than or equal to 80? true
Greater than or equal to 72? false
Greater than or equal to 76? false
Greater than or equal to 78? true
Greater than or equal to 77? false
Your number is 77
```

534

算法正确性证明。首先，我们必须证明算法的正确性：结果总能引向那个秘密的数字。我们通过如下事实加以证明：

- 区间始终包含秘密数字。
- 区间大小是 2 的整数次幂，从 n 开始递减。

第一条事实由代码保证其正确性；第二条事实指出，如果 $(hi-lo)$ 是 2 的幂，则 $(hi-lo)/2$ 是下一个较小的 2 的幂，并且得到两个减半的区间 $[lo, mid)$ 和 $[mid, hi)$ 。这些事实是利用归纳法证明算法操作正确性的基础。最终，区间的长度缩短为 1，所以我们可以保证查找到结果。

运行时间分析。假设 n 是可能值的个数。在程序 4.2.1 中，我们有 $n=2^k$ ，其中 $k=\lg n$ 。现在假设 $T(n)$ 是问题提问的次数。递归策略意味着 $T(n)$ 必须满足以下递推关系式：

$$T(n)=T(n/2)+1$$

当 $n=1$ 时， $T(1)=0$ 。把 n 替换为 2^k ，通过在这个式子上不断循环可以得到如下闭表达式：

$$T(2^k)=T(2^{k-1})+1=T(2^{k-2})+2=\cdots=T(1)+k=k$$

把 2^k 替换为 n （并且 k 替换为 $\lg n$ ）得到结果：

$$T(n)=\lg n$$

上述公式证明了我们的假设，使用二分查找算法的运行时间是对数型。请注意：即使 n 不是 2 的幂，二分查找算法和 `TwentyQuestions.binarySearch()` 也同样适用。我们假设 n 是 2 的幂只是为了简化证明的过程（参见练习 4.2.1）。

线性 - 对数之间的鸿沟。使用二分查找法的一种替代方法是先猜测 0，然后是 1、2、3，以此类推，直到找到那个秘密数字。我们称这个算法为顺序搜索。这是一个暴力算法的例子，虽然可以完成任务，但没有考虑成本。顺序搜索的运行时间与秘密数字相关：如果秘密数字为 0，顺序搜索只需要 1 步，但如果秘密数字是 $n-1$ ，顺序搜索则需要 n 步。如果随机选择数字，预期的步数为 $n/2$ 。同时，二分查找算法保证最多提出 $\lg n$ 个问题。 n 和 $\lg n$ 在实际应用中的差异是巨大的。理解这种巨大的差异是理解算法设计和分析重要性的关键步骤。在当前例子中，假设处理一个问题需要 1 秒。通过二分查找算法，你可以在 20 秒内猜出任何一个小于 100 万的数字；而使用顺序搜索的暴力算法，可能需要 100 万秒，超过一个星期的时间。我们将看到，在许多例子中，这种运行成本的差异会成为是否可以有效解决实际问题的决定因素。

[535]

二进制表示。如果我们重新回顾程序 1.3.7，就会立即发现二分查找算法几乎等同于将一个数字转换成二进制的计算方法！每个问题确定答案的一个二进制位。在我们的例子中，数字位于 0 到 127 之间的信息意味着其二进制表示的位数为 7，第一个问题（这个数字是否大于或等于 64？）的答案告诉我们第一个二进制位的值，第二个问题的答案告诉我们下一个二进制位的值，以此类推。例如，如果数值是 77，那么答案序列依次为“否是是否否是否”，可得出 1001101，即 77 的二进制表示。使用二进制表示法思考问题有助于理解对数 - 线性鸿沟：当程序的运行时间为参数 n 的线性函数时，其运行时间与 n 的值成正比，然而，对数运行时间仅仅与 n 的二进制位数成正比。换一种也许你更熟悉的场景，请考虑下面这个问题：你是想挣 6 美元还是 6 位数的薪水呢？

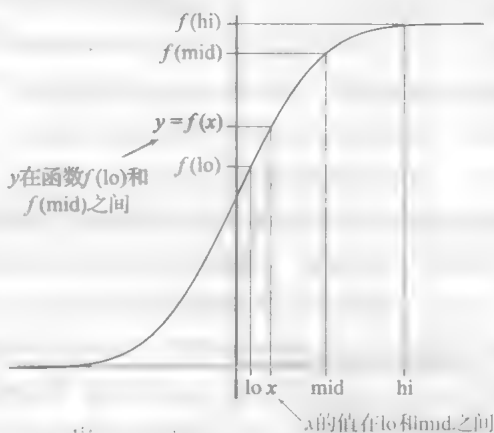
求反函数。作为二分查找法在科学计算中的一个应用实例，我们重新回顾计算递增函数 $f(x)$ 的反函数的问题。给定一个值 y ，我们的任务是找到一个值 x ，使得 $f(x)=y$ 。在这种

[536]

情况下，我们使用实数而不是整数作为区间的端点，但使用与猜测秘密数字相同的算法：每一步把区间长度减半，并确保 x 在区间中，直到区间足够小，此时值 x 位于期望的精度 δ 内。我们从包含 x 的区间 (lo, hi) 开始，使用如下递归策略：

- 计算中间值 $mid = lo + (hi - lo) / 2$ 。
- 基础步骤：如果 $hi - lo < \delta$ ，则返回 mid 作为 x 的估计值。
- 归纳步骤：否则，测试 $f(mid) > y$ 是否成立。如果成立，在 (lo, mid) 中查找 x ；否则在 (mid, hi) 中查找 x 。

为了解决这个问题，程序 4.2.2 计算了高斯累积分布函数 Φ 的反函数，我们在 Gaussian（程序 2.1.2）中实现了这个函数。



使用二分查找法求一个递增函数的反函数

程序 4.2.2 二分查找法

```
public static double inverseCDF(double y)
{ return bisectionSearch(y, 0.00000001, -8, 8); }

private static double bisectionSearch(double y, double delta,
double lo, double hi)
{ // 计算出cdf(x)=y时x的值
  double mid = lo + (hi - lo) / 2;
  if (hi - lo < delta) return mid;
  if (cdf(mid) > y)
    return bisectionSearch(y, delta, lo, mid);
  else
    return bisectionSearch(y, delta, mid, hi);
}
```

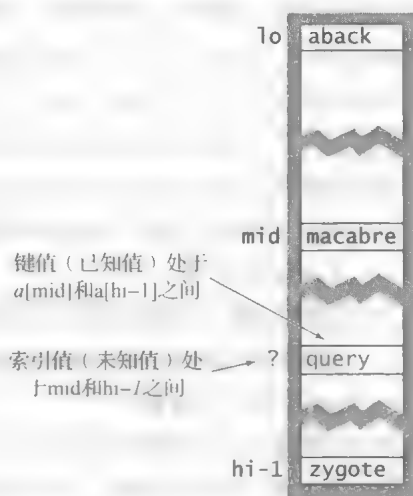
y	参数
delta	求解精度
lo	可能的最小值
mid	中点
hi	可能的最大值

对于我们的高斯函数库（程序 2.1.2），inverseCDF() 函数使用二分查找算法计算一个点 x ，使得 $\Phi(x)$ 等于给定值 y （误差精度为给定的 δ ）。它是一个递归函数，每次将包含点 x 的区间长度减半，求解位于区间中点的函数值，并利用 Φ 是递增函数的性质，判断所期望的点 x 位于左半区间还是右半区间，重复上述过程直到区间的长度小于给定的精度。

这个方法的关键在于函数是递增函数，即对于任何值 a 和 b ，如果 $f(a) < f(b)$ ，则 $a < b$ ，反之亦然。递归步骤仅仅运用了以下知识：如果已知 $y = f(x) < f(mid)$ ，则 $x < mid$ ，所以 x 一定位于区间 (lo, mid) 中，如果已知 $y = f(x) > f(mid)$ ，则 $x > mid$ ，所以 x 一定位于区间 (mid, hi) 中。这个问题可以被视为确定区间 (lo, hi) 的 n ($n = (hi - lo) / \delta$) 个长度为 δ 的小区间中，哪一个区间包含 x ，其运行时间是 n 的对数。与把一个整数转换为二进制一样，我们每次迭代确定 x 的一个位。在这种情况下，二分查找通常称为折半查找（bisection search），因为每一步我们都把区间一分为二。

在排好序的数组中使用二分查找法。二分查找法最重要的应用之一是使用一个关键字查找信息。这种用法在现代计算中无处不在，而依赖于相同概念的出版物正在走向消亡。

例如，在过去的几个世纪里，人们使用一个称为字典的出版物来查找单词的定义。在 20 世纪的大部分时间里，人们使用称为电话簿的出版物来查找一个人的电话号码。在这两种情形中，其基本机制都是相同的：所有的项目按顺序排列，通过可标识项目的关键字排序（例如字典中的单词、电话簿中的姓名，它们都按字母顺序排列）。你现在也许会使用计算机来查找这些信息，但是否考虑过如何在一本字典中查找一个单词？一种方法是顺序搜索，这是一种暴力算法，从头开始依次检查每个元素，直到找到该单词。没有人会使用这种算法。替代方法是打开书中间的某一页，在该页中查找该单词。如果查到则完成任务，否则，继续在本页前面的一半或本页后面的一半中重复上述查找过程。我们现在认识到这种方法就是二分查找法（程序 4.2.3）。



在排好序的数组中使用二分查找法（一步）

程序4.2.3 二分查找（在一个排好序的数组中）

```
public class BinarySearch
{
    public static int search(String key, String[] a)
    { return search(key, a, 0, a.length); }

    public static int search(String key, String[] a, int lo, int hi)
    { // 在a[lo, hi)区间中查找键值
      if (hi <= lo) return -1;
      int mid = lo + (hi - lo) / 2;
      int cmp = a[mid].compareTo(key);
      if (cmp > 0) return search(key, a, lo, mid);
      else if (cmp < 0) return search(key, a, mid+1, hi);
      else return mid;
    }

    public static void main(String[] args)
    { // 从标准输入获得键值，如果键值没有出现在
      // args[0]指定的文件中，那么就把它打印出来
      In in = new In(args[0]);
      String[] a = in.readAllStrings();
      while (!StdIn.isEmpty())
      {
          String key = StdIn.readString();
          if (search(key, a) < 0) StdOut.println(key);
      }
    }
}
```

key	查找关键字
$a[lo, hi)$	已排序的子数组
lo	最小索引
mid	中间索引
hi	最大索引

该类中的search()方法使用二分查找法查找一个位于排好序的数组中字符串key的索引（如果关键字不在数组中，则返回-1）。测试客户程序是一个异常过滤器，从一个命令行参数指定的文件中读取一个已排序的字符串数组（白名单列表），然后从标准输入获取一系列单词，并输出不包含在白名单中的单词。

```
% more emails.txt
bob@office
carl@beach
marvin@spam
bob@office
bob@office
mallory@spam
dave@boat
eve@airport
alice@home
```

```
% more whitelist.txt
alice@home
bob@office
carl@beach
dave@boat
```

```
% java BinarySearch whitelist.txt < emails.txt
marvin@spam
mallory@spam
eve@airport
```

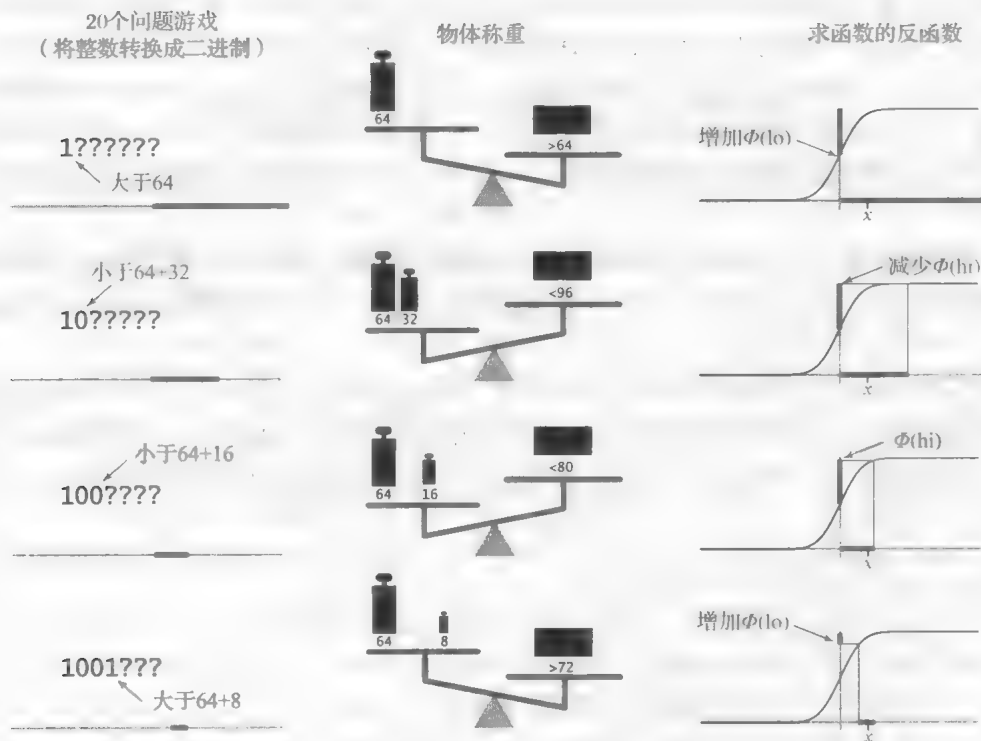
异常过滤器。我们将在 4.3 节中详细讨论实现替代字典或电话簿的计算机程序。程序 4.2.3 使用二分查找法解决一个简单的问题：一个给定的关键字是否存在于一个排好序的关键字数组中。例如，当检查一个单词的拼写时，只需要知道单词是否在字典中，而对单词的含义不感兴趣。在计算机搜索中，我们把信息保存在一个数组中，并按关键字排序（对于某些应用程序，其信息已经按顺序排列；对于其他程序中的信息，我们必须先使用本节随后讨论的算法对其进行排序）。

程序 4.2.3 中的二分查找算法与其他应用程序的区别主要有如下两点。首先，数组长度 n 不必是 2 的幂。其次，它必须考虑到所寻求的关键字可能不在数组中。当使用二进制编码来解释搜索过程时需要谨慎处理这些细节，我们在本节的问答环节和练习中进行讨论。

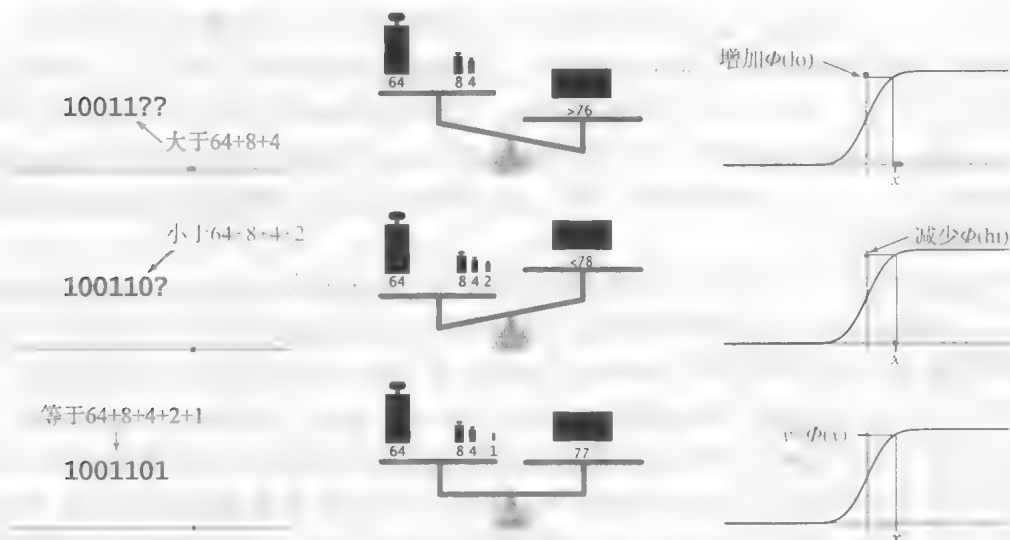
程序 4.2.3 中的测试客户程序被称为异常过滤器：它从一个文件读取一个有序的字符串数组（我们称之为白名单），并从标准输入中读取任意字符串序列，然后输出不在白名单中的字符串。异常过滤器具有很多直接的应用。例如，如果白名单是字典中的单词，而标准输入是一个文本文档，则异常过滤器会输出拼写错误的单词。另一个例子来自 Web 应用程序：电子邮件应用程序可能会使用异常过滤器来拒收任何不在白名单（包含你所有朋友的电子邮件地址）中的电子邮件。或者，操作系统可能使用异常过滤器禁止不在预先批准的 IP 地址白名单上的设备连接到你的计算机。

物体称重法。二分查找法自古以来就被人们所熟知，也许部分原因是以下应用场景。假设需要仅使用一个天平和一些砝码来确定一个物体的质量。使用二分查找，可以使用质量为 2 的幂的砝码（每个砝码只需要一个）做到这一点。把物体放在天平的右侧，左侧按照降序放上不同质量的砝码。如果一个砝码导致天平向左倾斜，则取出该砝码；否则，保留该砝码继续尝试。这个过程等价于通过递减 2 的幂的方法确定数字的二进制表示，如程序 1.3.7 中所示。

540



二分查找法的三个应用



二分查找法的三个应用 (续)

快速算法是现代世界的基本要素，二分查找法是描述快速算法应用效果的典型示例。通过一些简单的计算你将发现，对于使用异常过滤器在文档中查找所有拼写错误的单词或保护计算机免遭入侵之类的问题，你都需要一个类似于二分查找法的快速算法。花时间实现这样的快速算法，可以瞬间完成对一个百万单词文件的拼写检查，白名单也可以包含数百万单词，而使用暴力算法则可能需要几天或几周的时间。如今，互联网公司日常提供的服务一般基于二分查找算法，对于在包含数十亿元素的数组中实施数十亿次的查找操作，如果没有类似二分查找的快速算法，我们无法提供这样的服务。

无论是大量的实验数据还是对物理世界某些方面的具体表示，现代科学家都被淹没在大量的数据之中。二分查找法和类似的快速算法是科学进步的重要组成部分。使用暴力算法恰好类似于从第一页开始逐页查找字典中的单词。使用快速算法，你可以在瞬间搜索数十亿条信息。投入一些时间和精力来确定和使用这样一个快速的搜索算法，可以看出快速算法与暴力算法在问题的难易程度和花费资源的大小上的不同，甚至暴力算法可能根本无法解决你的问题。

插入排序算法 二分查找法要求数组必须是排好序的，排序还有许多其他直接的应用，所以我们接下来讨论排序算法。我们首先讨论暴力算法，然后讨论可以应用于大型数组的复杂算法。

我们讨论的暴力算法称为插入排序算法，该算法基于人们用来排列一手扑克的基本方法，即每次处理一张牌，然后将它插入已经排好序的扑克牌中，使它们保持按序排列。以下代码是在 Java 中模拟这一过程，该方法可以重新排列数组中的字符串，使其按升序排列：

```
public static void sort(String[] a)
{
    int n = a.length;
    for (int i = 1; i < n; i++)
        for (int j = i; j > 0; j--)
            if (a[j-1].compareTo(a[j]) > 0)
                exchange(a, j-1, j);
            else break;
}
```

外层循环的每次循环开始时，数组中的前 i 个元素是按顺序排列的，内层循环将 $a[i]$ 移动到数组中的正确位置，如下面的示例中，当 i 是 6 时：

i	j	a[]							
		0	1	2	3	4	5	6	7
6	6	and	had	him	his	was	you	the	but
6	5	and	had	him	his	was	the	you	but
6	4	and	had	him	his	the	was	you	but
		and	had	him	his	the	was	you	but

通过将 $a[6]$ 与左边更大的数字交换，使 $a[6]$ 被插入数组中合适的位置

具体来说，通过与左边较大元素的交换操作（使用 2.1 节中实现的 `exchange()` 函数），元素 $a[i]$ 被依次移动插入其左侧已排序元素的前面，逐一从右向左移动，直到元素到达合适的位置。表格中展示的三行就是程序运行时的跟踪信息，注意黑色的元素就是用于比较的元素。

上述插入过程首先针对 i 等于 1 的情况执行，然后 i 等于 2、3，详细的跟踪信息如下所示。 543

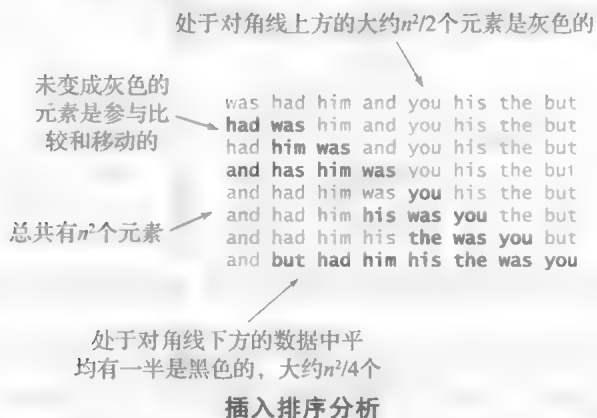
i	j	a[]								
		0	1	2	3	4	5	6	7	
		was	had	him	and	you	his	the	but	
1	0	had	was	him	and	you	his	the	but	
2	1	had	him	was	and	you	his	the	but	
3	0	and	had	him	was	you	his	the	but	
4	4	and	had	him	was	you	his	the	but	
5	3	and	had	him	his	was	you	the	but	
6	4	and	had	him	his	the	was	you	but	
7	1	and	but	had	him	his	the	was	you	
		and	but	had	him	his	the	was	you	

插入排序的跟踪信息（将 $a[1] \sim a[n-1]$ 插入合适位置）

跟踪信息显示了每次外层的 `for` 循环完成后的数组内容，以及这个时间点 j 的值。加阴影显示的元素是循环开始时 $a[i]$ 的值，黑色的元素是 `for` 循环中涉及交换和移动到右侧一个位置的元素。对于每一个 i 值，当循环结束时，元素 $a[0]$ 到 $a[i-1]$ 是按顺序排列的。当 i 为 $a.length$ 时，最后一次循环完成。上述讨论再次说明了在研究或开发新算法时需要做的第一件事就是：证明算法的正确性。这样做提供了研究算法性能和有效使用算法的基础。

运行时间分析。插入排序的数据交换部分代码处于双重嵌套的 `for` 循环内部，这表明运行时间是二次型的，但是由于 `break` 语句的存在，不能立刻得出这个结论。例如，在最理想的情况下，也就是输入数组已经是按序排列的时候，内层 `for` 循环仅仅需要进行一次比较（对于每个从 1 到 $n-1$ 的 j ，判断 $a[j-1]$ 是否小于或等于 $a[j]$ ），若是则结束循环），所以总运行时间是线性的。如果输入数组是逆序排序，则内层循环会全部执行直到结束都不会中断，所以，内层循环指令的执行次数为 $1+2+\cdots+n-1 \sim \frac{1}{2}n^2$ ，即运行时间是二次的。为了理解插入排序对于随机排序数组的性能，仔细观察运行跟踪信息：这是一个 $n \times n$ 数组，黑色元素对应于每次交换。也就是说，黑色元素的个数是内层循环中比较和交换指令执行的次数。我们预计每个新插入元素可能被插入到任何位置，所以平均来说，元素将向左移动一半距离。因此，我们预计平均情况下对角线下面一半的元素（总共约 $n^2/4$ ）是黑色的。这样，可以直接 544

假设：针对随机排序的输入数组，插入排序的预期运行时间是二次型。



排序其他类型的数据。我们希望能够排序所有类型的数据，而不仅仅是字符串。在科学应用程序中，我们也许希望通过数值对实验结果进行排序；在商业应用程序中，我们可能希望使用货币金额、时间或日期；在系统软件中，我们可能希望使用 IP 地址或进程 ID。在以上这些应用场景中排序的想法是很自然的，但是实现一种适用于所有情况的排序方法就需要提供抽象机制，像 Java 接口提供的功能一样。为了排序数组中的对象，我们只需要假设可以通过某种机制比较两个元素，以查看第一个元素与第二个元素是大于、小于还是等于关系。Java 为此提供了 `java.util.Comparable` 接口。

```
public interface Comparable<Key>
```

```
    int compareTo(Key b) 将这个对象与对象b进行比较，以便对二者排序
```

Java 中 `java.util.Comparable` 接口的 API

实现 `Comparable` 接口的类为其对象实现 `compareTo()` 方法，`a.compareTo(b)` 的功能是在 `a` 小于 `b` 时返回负整数（通常为 `-1`），在 `a` 大于 `b` 时返回正整数（通常为 `+1`），在 `a` 等于 `b` 时返回 `0`（我们将在 4.3 节中介绍 `<Key>` 符号，它用来确保被比较的两个对象具有相同的类型）。

[545] 小于、大于、等于的确切含义取决于数据类型，不尊重关于这些概念的自然数学法则的实现将产生无法预知的结果。更正式地说，`compareTo()` 方法必须定义一个总顺序。这意味着以下三个属性必须成立（我们使用符号 $x \leq y$ 作为 `x.compareTo(y) <= 0` 的缩写，以及 $x=y$ 作为 `x.compareTo(y) == 0` 的简写）：

- 反对称性：如果 $x \leq y$ 并且 $y \leq x$ ，则 $x=y$ 。
- 传递性：如果 $x \leq y$ 并且 $y \leq z$ ，则 $x \leq z$ 。
- 总体有序： $x \leq y$ 或 $y \leq x$ 至少有一个成立，或者两者都成立。

这三个属性适用于各种常见的排序，包括字符串的字母顺序和整数、实数的升序。我们将实现 `Comparable` 接口的数据类型称为可比较的数据类型，并将相关的总顺序作为其自然顺序。Java 的 `String` 类型是可比较的，与 3.3 节介绍的基本封装类型（如 `Integer` 和 `Double`）类似。

按照这个约定，`Insertion`（程序 4.2.4）实现了一个排序方法，将一个可比较的对象作为参数，并按照 `compareTo()` 指定的顺序重新排列数组，使其元素按照升序排列。现在，我们可以使用 `Insertion.sort()` 对 `String[]`、`Integer[]` 或 `Double[]` 类型的数组进行排序。

使数据类型具有可比性也很容易，以便我们可以对用户定义的数据类型进行排序。为了实现这一点，必须在类声明中添加 `implements Comparable` 语句，然后添加一个 `compareTo()` 方法来定义总顺序。例如，为了使 `Counter` 数据类型具有可比性，我们如下修改程序 3.3.2：

```
public class Counter implements Comparable<Counter>
{
    private int count;
    ...
    public int compareTo(Counter b)
    {
        if (count < b.count) return -1;
        else if (count > b.count) return +1;
        else return 0;
    }
    ...
}
```

现在，我们可以使用 `Insertion.sort()` 按升序对 `Counter` 对象数组进行排序。

546

程序4.2.4 插入排序

```
public class Insertion
{
    public static void sort(Comparable[] a)
    {
        // 将a[]中的元素按升序排列
        int n = a.length;
        for (int i = 1; i < n; i++)
            // 将a[i]插入正确的位置
            for (int j = i; j > 0; j--)
                if (a[j].compareTo(a[j-1]) < 0)
                    exchange(a, j-1, j);
                else break;
    }

    public static void exchange(Comparable[] a, int i, int j)
    { Comparable temp = a[j]; a[j] = a[i]; a[i] = temp; }

    public static void main(String[] args)
    { // 从标准输入获得一系列字符串，对它们进行排序并打印出来
      String[] a = StdIn.readAllStrings();
      sort(a);
      for (int i = 0; i < a.length; i++)
          StdOut.print(a[i] + " ");
      StdOut.println();
    }
}
```

a[]	待排序数组
n	数组长度

`sort()`函数是插入排序的一个实现。它对实现`Comparable`接口的任何类型的数据进行排序（因此，有一个`compareTo()`方法）。`Insertion.sort()`仅适用于小数组或几乎处于有序状态的数组，其对于乱序的大型数组来说，排序过程太慢了。

```
% more 8words.txt
was had him and you his the but
% java Insertion < 8words.txt
and but had him his the was you
% java Insertion < TomSawyer.txt
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick
```

547

实证分析。`InsertionDoublingTest`（程序 4.2.5）通过在 n 个随机 `Double` 对象上运行 `Insertion.sort()`，并计算其运行时间的比率来测试我们的假设：对于随机排序的数组，插入排序是二次型的。这个比率收敛到 4，验证了插入排序运行时间是二次型的假设，正如上一节所述。如

果读者在自己的计算机上运行 InsertionDoublingTest, 将会理解得更深刻。你可能会注意到对于不同的 n 值, 系统的缓存或其他机制会对性能有不同的影响, 但运行时间是二次型的这个结论应该十分明显, 所以你很快就会相信插入排序的运行速度太慢, 无法适用于大型输入的情况。

对输入的敏感性。请注意, InsertionDoublingTest 有一个命令行参数 trials, 并针对每个问题规模进行 trials 次实验, 而不是一次。正如我们所观察到的结果, 这样做的原因之一是插入排序算法的运行时间对输入值敏感。这与诸如 ThreeSum 程序的行为完全不同, 所以我们需要认真解释分析结果。断然预测插入排序算法的运行时间是二次型的不合适, 因为应用程序可能包含运行时间为线性的输入情况。当一个算法的性能对输入值敏感时, 如果不考虑输入因素则可能无法做出准确的预测。

存在很多自然的应用场景, 对它使用插入排序时算法的运行时间可能是二次型的, 所以我们需要考虑更快速的排序算法。正如本书 4.1 节所述, 大致的计算结果表明使用一台速度更快的计算机并没有太大的帮助。字典、科学数据库或商业数据库可能包含数以十亿计的元

548

程序4.2.5 插入排序算法的倍增测试

```
public class InsertionDoublingTest
{
    public static double timeTrials(int trials, int n)
    { // 对包含n个随机元素的数组进行排序
      double total = 0.0;
      Double[] a = new Double[n];
      for (int t = 0; t < trials; t++)
      {
          for (int i = 0; i < n; i++)
              a[i] = StdRandom.uniform(0.0, 1.0);
          Stopwatch timer = new Stopwatch();
          Insertion.sort(a);
          total += timer.elapsedTime();
      }
      return total;
    }

    public static void main(String[] args)
    { // 打印插入排序在输入规模倍增时的时间增长系数
      int trials = Integer.parseInt(args[0]);
      for (int n = 1024; true; n *= 2)
      {
          double prev = timeTrials(trials, n/2);
          double curr = timeTrials(trials, n);
          double ratio = curr / prev;
          StdOut.printf("%7d %4.2f\n", n, ratio);
      }
    }
}
```

trials	n	total	timer	a[]	prev	curr	ratio
实验运行次数	问题规模	消耗的总时间	计时器	待排序数组	长度为n/2的数组排序的时间	当前实验的运行时间	当前实验倍增的比率

timeTrials()方法针对随机排列的double型数组运行Insertion.sort()函数。第一个参数n是数组的长度, 第二个参数trials是运行的次数。多次实验会产生更精确的结果, 因为多次实验可以减少系统影响和对输入的依赖。

```
% java InsertionDoublingTest 1
1024 0.71
2048 3.00
4096 5.20
8192 3.32
16384 3.91
32768 3.89
```

```
% java InsertionDoublingTest 10
1024 1.89
2048 5.00
4096 3.58
8192 4.09
16384 4.83
32768 3.96
```

549

归并排序 为了开发一个更快的排序方法，我们使用递和分治的方法来设计算法，希望每个程序员都能理解这种算法。分治法是指解决问题的一种方法或思想，即把问题分解成独立的部分，然后独立地解决它们，最后使用不同部分的解决方案形成一个总的解决方案。为了使用这个策略对一个数组进行排序，我们把数组分成两部分，分别对这两部分进行排序，然后合并结果从而对整个数组进行排序。这个算法被称为归并排序算法。

处理一个给定数组的连续子数组时，我们使用 $a[lo,hi]$ 表示 $a[lo], a[lo+1], \dots, a[hi-1]$ (采用与二分查找算法中使用的相同规则表示半开区间，其不包含 $a[hi]$)。为了排序 $a[lo, hi]$ ，我们使用如下递归策略：

- 基础步骤：如果子数组的长度为 0 或 1，则排序完成。
- 归纳步骤：否则，计算 $mid=lo+(hi-lo)/2$ ，递归地对两个子数组 $a[lo, mid]$ 和 $a[mid, hi]$ 排序并合并它们。

输入
was had him and you his the but
对左边一半进行排序
and had him was
对右边一半进行排序
but his the you
合并
and but had him his the was you
归并排序概述

i	j	k	aux[k]	a[]							
				0	1	2	3	4	5	6	7
				and	had	him	was	but	his	the	you
0	4	0	and	and	had	him	was	but	his	the	you
1	4	1	but	and	had	him	was	but	his	the	you
1	5	2	had	and	had	him	was	but	his	the	you
2	5	3	him	and	had	him	was	but	his	the	you
3	5	4	his	and	had	him	was	but	his	the	you
3	6	5	the	and	had	him	was	but	his	the	you
3	7	6	was	and	had	him	was	but	his	the	you
4	7	7	you	and	had	him	was	but	his	the	you

将已排序的左子数组与已排序的右子数组合并的跟踪信息

550

Merge (程序 4.2.6) 是这个算法的一种实现。数组元素通过递归调用代码重新排列，然后合并递归调用后数组的两个部分。通常，理解归并过程最简单的方法是分析一次排序过程的跟踪信息。代码中索引 i 用于第一个子数组，索引 j 用于第二个子数组，第三个索引 k 用于存储结果的辅助数组 $aux[]$ 。归并的实现是一个单循环，将 $aux[k]$ 设置为 $a[i]$ 或 $a[j]$ (然后递增 k ，以及 i 和 j 中被采用的那个值)。如果 i 或 j 中的任何一个已经到达子数组的末端，则用另一个子数组填充 $aux[k]$ 剩下的空间。否则，将 $aux[k]$ 设置为 $a[i]$ 或 $a[j]$ 中较小的一个。当两个子数组中的所有元素都复制到 $aux[]$ 后，将 $aux[]$ 中的排序结果复制给原始数组。请读者务必花点时间来研究给出的跟踪信息，以更好地理解代码为什么总是可以正确合并两个排好序的子数组，从而排序整个数组的原理。

递归方法可以保证两个子数组在合并前正确排序。同样，理解这个过程的最好方法是研究每次递归调用 $sort()$ 方法后返回的数组内容的跟踪信息。下面显示了我们这个例子的跟踪信息。首先合并 $a[0]$ 和 $a[1]$ 为一个排序子数组 $a[0, 2)$ ，然后合并 $a[2]$ 和 $a[3]$ 为排序子数组 $a[2, 4)$ ，然后将这两个大小为 2 的子数组合并为一个排序子数组 $[0, 4)$ ，以此类推。如果你已经确信了合并处理的正确性，则只需要理解若代码可以正确地分割数组，那么它就能够准确地排序。请注意，当要排序的数组中元素数量不是偶数时，左半部分会比右半部分少一个元素。

	a[]							
	0	1	2	3	4	5	6	7
	was	had	him	and	you	his	the	but
sort(a, aux, 0, 8)								
sort(a, aux, 0, 4)								
sort(a, aux, 0, 2)								
return	had	was	him	and	you	his	the	but
sort(a, aux, 2, 4)								
return	had	was	and	him	you	his	the	but
return	and	had	him	was	you	his	the	but
sort(a, aux, 4, 8)								
sort(a, aux, 4, 6)								
return	and	had	him	was	his	you	the	but
sort(a, aux, 6, 8)								
return	and	had	him	was	his	you	but	the
return	and	had	him	was	but	his	the	you
return	and	but	had	him	his	the	was	you

使用递归调用进行归并排序的跟踪

程序4.2.6 归并排序

```

public class Merge
{
    public static void sort(Comparable[] a)
    {
        Comparable[] aux = new Comparable[a.length];
        sort(a, aux, 0, a.length);
    }

    private static void sort(Comparable[] a, Comparable[] aux,
                             int lo, int hi)
    {
        // 对 a[lo, hi) 排序
        if (hi - lo <= 1) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid, hi);
        int i = lo, j = mid;
        for (int k = lo; k < hi; k++)
            if (i == mid) aux[k] = a[j++];
            else if (j == hi) aux[k] = a[i++];
            else if (a[j].compareTo(a[i]) < 0) aux[k] = a[j++];
            else aux[k] = a[i++];
        for (int k = lo; k < hi; k++)
            a[k] = aux[k];
    }

    public static void main(String[] args)
    {
        /* 见程序 4.2.4. */
    }
}

```

a[lo, hi)	待排序的子数组
lo	最小索引
mid	中间索引
hi	最大索引
aux[]	辅助数组

sort()函数是归并排序的一个实现。它对实现Comparable接口的任何数据类型的数组进行排序。与Insertion.sort()相比, 这个实现适用于排序数据量很大的数组。

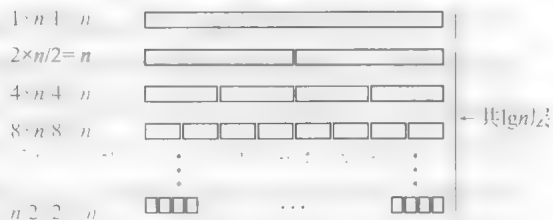
```

% java Merge < 8words.txt
was had him and you his the but

% java Merge < TomSawyer.txt
... achievement aching aching acquire acquired ...

```


运行时间分析。归并排序的内层循环以辅助数组为中心。两个 for 循环都包含 n 次迭代，因此内层循环中指令的执行频率与递归函数调用的子数组长度之和成正比。当我们按照子数组的大小对调用过程进行分层时，就很容易观察指令执行的总量。为了方便起见，我们假设 n 是 2 的幂，即 $n=2^k$ 。在第一层，包括一个大小为 n 的调用；在第二层，包括两个大小为 $n/2$ 的调用；在第三层，包括四个大小为 $n/4$ 的调用。以此类推，直到 $n/2$ 调用大小为 2 时，就是最后一层调用。完成所有的计算正好有 $k=\lg n$ 层调用，因此归并排序算法的循环中指令执行的总次数为 $n \lg n$ 。这个公式也证明了归并排序算法的运行时间是线性对数的假设。请注意：当 n 不是 2 的幂时，每一层的子数组的大小不一定是相同的，但层的数目仍然是对数型的，所以线性对数假设对所有的 n 都是成立的（参见练习 4.2.18 和 4.2.19）。



归并排序内层循环执行次数计数（当 n 是 2 的幂时）

建议你在自己的计算机上运行像程序 4.2.5 中的 Merge.sort() 来测试前面的代码。尝试运行后，你将会发现 Merge.sort() 比 Insertion.sort() 要快得多，因而可以相对容易地排序超大型的数组。验证运行时间是线性对数型的假设需要更多的工作，但你可以看到归并排序可以解决那些使用暴力破解算法（如插入排序）无法解决的问题。

二次型与线性对数型之间的鸿沟。 n^2 和 $n \lg n$ 之间的差异导致这两个算法在实际应用中有很大的不同，就像通过二分查找算法克服线性 - 对数之间的鸿沟一样。了解这种巨大的差异是理解算法设计和分析重要性的另一个关键步骤。对于许多重要的计算问题，从二次型到线性对数型的速度提升（比如通过归并排序实现排序）会使得程序能够解决涉及大量数据的问题，而这些问题使用二次型的算法可能根本无法有效解决。

553

分治算法。我们使用的分治方法对于许多重要问题都是有效的，如果你学习了算法设计课程的话，将会了解到这一点。目前，我们鼓励你研究本节结尾的部分练习题，这些练习描述了分治算法能够解决的一系列问题，而没有分治算法，这些问题根本就不可能解决。

归约为排序问题。如果我们可以用问题 B 的解决方案来解决问题 A，则称问题 A 可归约为问题 B。从零开始设计一个新的分治算法有时类似于解决一个需要一定经验和独创性的难题，所以你可能没有信心解决这个问题。然而一个简单的方法通常是有效的：给定一个新的问题，问问自己如果数据进行排序后将如何解决。通常情况下，针对一个排好序的数组，一个相对简单的线性扫描就可以完成任务。对于这类情况，利用类似归并排序算法中的智慧，我们就得到了线性对数型算法。例如，确定一个数组中的每个元素是否都是不同的，也被称为元素不同性问题，就可以归约为排序，因为我们可以对数组进行排序，然后通过扫描排好序的数组来检查一个元素的值是否等于下一个，如果都不相等，那么值是不同的。再如，实现 StdStats.median()（参见 2.2 节）的简单方法是将选择归约为排序。接下来我们分析一个更复杂的例子，你也可以在本节最后的练习中找到许多其他例子。

归并排序算法归功于物理学家约翰·冯·诺依曼（John von Neumann），他是最早认识到计算在科学研究中重要性的科学家之一。冯·诺依曼为计算机做出了许多贡献，包括自 20 世纪 50 年代使用至今的计算机体系结构的基本概念。关于应用程序编程，冯·诺依曼提出了如下观点：

- 排序是许多应用程序的基本组成部分。

- 二次型算法对于许多实际问题来说太慢了。
- 分治算法是有效的。
- 证明程序的正确性并知道其开销非常重要。

虽然计算机的速度比以前要快很多个数量级，而且比冯·诺依曼可用的计算机有更多的内存量，但是这些基本概念在今天依然很重要。有效且成功使用计算机的人都明白，在通常情况下暴力算法是不足以解决问题的，这个局面和冯·诺依曼的时代一样。

[554]

应用程序：频率计数 FrequencyCount (程序 4.2.7) 从标准输入中读取字符串序列，向标准输出写入查找到的不同字符串以及出现的次数，并按照频率降序排列。这种计算适用于许多应用：一个语言学家可能正在研究文本中单词的使用模式，一个科学家可能正在实验数据中寻找经常发生的事件，一个商人可能正在寻找最常出现在交易列表中的客户，一个网络分析师可能正在寻找最活跃的用户。每个应用程序都可能包含数百万或者更多个字符串，所以我们需要一个线性对数算法（或更好的算法）。FrequencyCount 是通过将问题归约为排序来找到解决问题的算法。实际上，这个程序进行了两次排序操作。

计算频率。第一步是从标准输入读取字符串并将其排序。在这种情况下，我们感兴趣的不是字符串是否按照字母序排列，而是排序操作使得相同的字符排列在一起。如果输入是

```
to be or not to be to
```

那么排序的结果是

```
be be not or to to to
```

相同的字符串（例如在数组中 be 出现了两次、to 出现了三次）在数组中排列在一起。所以扫描数组一遍就可以完成频率的计算。我们在 3.3 节中提到的 Counter 数据类型是完成这项工作的完美工具。回想一下，Counter (程序 3.3.2) 包含一个字符串实例变量（初始化为构造函数参数）、一个计数实例变量（初始化为 0）和一个使计数器递增 1 的 increment() 实例方法。我们维护一个整数 m 和一个 Counter 对象数组 zipf[]，并针对每个字符串执行以下操作：

- 如果字符串不等于前一个字符串，则创建一个新的 Counter 对象并将 m 加 1。
- 把最近创建的 Counter 对象加 1。

[555]

最后，m 的值是不同字符串的数量，zipf[i] 包含第 i 个字符串及其频率。

排序频率。接下来，我们按照频率对 Counter 对象进行排序。

可以在客户程序代码中这样做的前提是 Counter 实现了 Comparable 接口，其 compareTo() 方法通过 count 来比较对象（请参阅练习 4.2.14）。一旦完成，我们只需对数组进行排序即可！请注意，FrequencyCount 中按照最大可能长度申请了 zipf[] 数组，而我们只需对其子数组进行排序。其实我们可以在分配 zipf[] 之前对 words[] 进行一次扫描，以确定不同字符串的数量。我们把修改和优化 Merge (程序 4.2.6) 的工作留作练习（参见练习 4.2.15）。

齐普夫定律 (Zipf's law)。程序 FrequencyCount 最常见的应用是基本语言分析：哪些词在文本中出现频率最高？通过现象观测表

i	M	a[i]	zipf[i].value()			
			0	1	2	3
	0					
0	1	be	1			
1	1	be	2			
2	2	not		1		
3	3	or			1	
4	4	to				1
5	4	to				2
6	4	to				3
			2	1	1	3

频率计数

i	zipf[i]	
之前		
0	2	be
1	1	not
2	1	or
3	3	to
之后		
0	1	not
1	1	or
2	2	be
3	3	to

频率排序

明，在一篇包含 m 个不同单词的文章中，第 i 个最频繁出现单词的频率正比于 $1/i$ ，其比例常数是谐波数 H_m 的倒数，这被称为齐普夫定律。例如，第二个最常出现的单词的出现频率是第一个的一半左右。令人惊讶的是，这个经验假设可以适用于各种场景，从财务数据到网络使用统计数据。程序 4.2.7 中的测试客户程序使用了一个包含从网络中随机抽取的 100 万个语句的数据库，用以验证齐普夫定律（请参见本书官网）。

你很可能会发现，在编写程序来完成一个简单任务时，先对数据进行排序可以帮助你很容易地解决这个问题。有多少不同的值？哪个值最常出现？中间值是什么？使用归并排序等线性对数型排序算法，甚至可以解决巨型数据集的问题。FrequencyCount 就是一个很好的例子，它使用了两次不同类型数据的排序操作。如果暴力排序不适用，则可能会应用其他一些分治算法，或者可能需要一些更复杂的方法。如果没有一个好的算法（以及对其性能特征的理解），你可能会觉得很沮丧：为什么自己快速和昂贵的计算机却无法解决看起来很简单的问题？随着问题规模的增大，如果你知道如何有效地解决这个问题，你会发现计算机可以成为一台超乎想象的计算工具。

556

程序4.2.7 频率计数

```
public class FrequencyCount
{
    public static void main(String[] args)
    { // 按照出现频率的降序排列输入的字符串

        String[] words = StdIn.readAllStrings();
        Merge.sort(words);
        Counter[] zipf = new Counter[words.length];
        int m = 0;
        for (int i = 0; i < words.length; i++)
        { // 创建新的计数器，或者将上一个计数器加1
            if (i == 0 || !words[i].equals(words[i-1]))
                zipf[m++] = new Counter(words[i], words.length);
            zipf[m-1].increment();
        }
        Merge.sort(zipf, 0, m);
        for (int j = m-1; j >= 0; j--)
            StdOut.println(zipf[j]);
    }
}
```

s 输入信息
words[] 输入中的字符串
zipf[] 计数器数组
m 不同字符串的个数

该程序对标准输入中的单词进行排序，使用排好序的数组计算每个单词的出现频率，然后对频率进行排序。下面使用的测试文件超过2000万字。在绘制的图中，柱状图用于表示第*i*个词与第一个词频率的比值，曲线表示 $1/i$ ，你可以看到它们的相对关系。

```
% java FrequencyCount < Leipzig1M.txt
the: 1160105
of: 593492
to: 560945
a: 472819
and: 435866
in: 430484
for: 205531
The: 192296
that: 188971
is: 172225
said: 148915
on: 147024
was: 141178
by: 118429
...
```



557

经验总结 我们可以通过编写程序来解决很多复杂的实际问题，而我们编写的绝大多数程序均需要开发清晰正确的解决方案，将程序分解成可控大小的模块，完成开发并测试和调试它们，最终给出解决方案。从一开始，我们就在本书中遵循这个原则和方法以进行程序设计。但是当你致力于开发更复杂的应用程序时，你会发现清晰正确的解决方案并不一定能解决所有的问题，因为计算成本可能是一个限制因素。本节中的例子就是对这个事实的基本阐述。

重视计算成本。如果你能用一个简单的算法很快解决一个小规模问题，那是最好的。但是，如果你需要解决涉及大量数据或大量计算的问题，则必须要考虑成本。

归约为一个已知问题。我们使用频率计数排序说明了理解基本算法并将其用于解决问题的效用。

分治算法。你应该思考一下分治的力量，正如开发线性对数型排序算法（归并排序）所表明的那样，该算法是解决很多计算问题的基础。分治只是开发高效算法的一种方法。

自从计算出现以来，人们就一直致力于开发可以高效解决问题的算法，如二分查找法和归并排序算法等。这个研究领域被称为“算法设计与分析”，其研究包括设计规范（如分治和动态规划等）、关于算法性能预估的技术、用于解决各种实际问题的算法（如排序和查找等）等。你可以在 Java 库或其他专业库中找到许多这些算法的实现，它们是我们计算的基本工具，但是理解这些工具就像理解数学或科学的基本工具一样。你可以使用矩阵处理包来求解矩阵的特征值，但是仍然需要学习线性代数课程。现在你已经知道一个快速算法可以产生徒劳无获和正确解决一个实际问题的差异，下面可以开始寻找在哪些情况下算法设计和分析的知识可以有所作为，以及在哪里使用高效的算法（如二分查找和归并排序）可以有效地解决问题。

[558]

问答环节

问：为什么我们需要花费如此巨大的精力来证明程序的正确性？

答：为了避免更大的痛苦。二分查找法就是一个著名的例子。你现在已经理解了二分查找法。一个经典的实现方法是使用 `while` 循环而不是递归。试着解决练习 4.2.2，但不要参考书中的代码。Jon Bentley 曾经做过一个著名的实验，要求几个专业程序员编写该程序，结果他们的解决方案大多数都是不正确的。

问：Java 库中是否实现了排序和查找？

答：是的。Java 包 `java.util` 中包含了实现归并排序和二分查找的静态方法 `Arrays.sort()` 和 `Arrays.binarySearch()`。实际上，每个方法代表着一系列同名的重载方法，分别用于 `Comparable` 类型和所有基本类型（每个类型一个）。

问：那么本书为什么不使用它们呢？

答：其实可以随意使用它们。但正如我们研究过的很多主题一样，如果你了解其背后的原理知识，就可以更有效地使用这些工具。

问：请解释为什么我们使用 $lo+(hi-lo)/2$ 来计算 `lo` 和 `hi` 的中点，而不是使用 $(lo+hi)/2$ 。

答：后者在计算 `lo+hi` 时可能会有 `int` 型溢出的错误。

问：编译 `Insertion.java` 和 `Merge.java` 时，为什么会出现“unchecked or unsafe operation”（未经检查或不安全的操作）警告？

答：sort() 的参数是一个 Comparable 类型的数组，但是在技术上并没有禁止使用不同类型元素的数组。要消除警告，请将签名更改为：

```
public static<Key extends Comparable<Key>>void sort(Key[] a)
```

我们将在下一节讨论泛型 (generic) 时学习 <Key> 符号。

559

练习

4.2.1 为 Questions (程序 4.2.1) 开发一个新版本的实现，从命令行接收参数 n 作为最大数字。证明你的实现是正确的。

4.2.2 实现 BinarySearch (程序 4.2.3) 的非递归版本。

4.2.3 修改 BinarySearch (程序 4.2.3)，实现如下功能：如果搜索的关键字在数组中存在，则返回满足 $a[i]$ 等于 key 的最小索引 i ，否则返回 $-i$ ，其中 i 是 $a[i]$ 比 key 大的最小索引。

4.2.4 如果将二分查找法应用于无序数组，描述会发生什么情况。为什么不在每次调用二分查找法之前检查数组是否有序？请问你是否可以判断二分查找算法检查的元素是不是升序排列的？

4.2.5 描述为什么二分查找算法需要使用不可变的关键字。

4.2.6 向 Insertion 中添加代码，以生成正文中给出的跟踪信息。

4.2.7 向 Merge 中添加代码以生成如下所示的跟踪信息：

```
% java Merge < tiny.txt
was had him and you his the but
had was
      and him
and had him was
      his you
      but the
      but his the you
and but had him his the was you
```

4.2.8 以正文中的样式，给出插入排序和归并排序的跟踪信息，假设输入如下：it was the best of times it was。

4.2.9 实现程序 4.2.2 更通用化的版本，将二分查找算法应用于任何单调递增的函数。使用与 3.3 节中数值积分示例相同的函数式编程。

560

4.2.10 编写一个过滤器程序 DeDup，它从标准输入中读取字符串，在标准输出中输出去掉所有重复字符串后的字符串序列（并保持排序状态）。

4.2.11 修改 StockAccount (程序 3.2.8)，使其实现 Comparable 接口（按名称比较股票账户）。提示：使用 String 数据类型中的 compareTo() 方法完成这项繁重的工作。

4.2.12 修改 Vector (程序 3.3.3)，为其实现 Comparable 接口，然后使用字典顺序按坐标比较向量。

4.2.13 修改 Time (练习 3.3.21)，为其实现 Comparable 接口，然后按时间先后顺序比较时间值。

4.2.14 修改 Counter (程序 3.3.2)，为其实现 Comparable 接口，然后通过频率计数比较对象。

4.2.15 在 Insertion (程序 4.2.4) 和 Merge (程序 4.2.6) 中添加方法以支持排序子数组。

4.2.16 开发非递归版本的归并排序算法 (程序 4.2.6)。为了简单起见，假设项的数目 n 是 2 的幂。加分题：即使 n 不是 2 的幂，也要使你的程序正常工作。

4.2.17 请统计你最喜欢的小说中单词的频率分布。请问结果遵守齐普夫定律吗？

4.2.18 从数学上分析使用归并排序处理长度为 n 的数组时比较操作发生的次数。为了简单起见，假设 n 是 2 的幂。

答案：设 $M(n)$ 为归并排序长度为 n 的数组的比较次数。合并两个总长度为 n 的子数组需要 $\frac{1}{2}n$ 到 $n-1$ 次比较。因此， $M(n)$ 满足以下递推关系：

$$M(n) \leq 2M(n/2) + n$$

当 $n=1$ 时， $M(1)=0$ 。用 2^k 代替 n 给出

$$M(2^k) \leq 2M(2^{k-1}) + 2^k$$

这与我们给出的二分查找法的递推公式相似，但更复杂。如果我们将公式两边同时除以 2^k ，就可以得到

$$M(2^k)/2^k \leq M(2^{k-1})/2^{k-1} + 1$$

这正好与二分查找法一致。也就是说， $M(2^k)/2^k \leq T(2^k) = n$ 。用 n 代替 2^k （并且 $\lg n$ 代替 k ），结果得到 $M(n) \leq n \lg n$ 。通过类似的证明可以得到 $M(n) \geq \frac{1}{2} n \lg n$ 。

4.2.19 分析 n 不是 2 的幂时的归并排序情况。

部分解答：当 n 是一个奇数时，一个子数组比另一个多一个元素，所以当 n 不是 2 的幂时，每一层的子数组不一定是相同的大小。同样，每个元素必定会出现在某一个子数组中，并且层的数量仍然是对数型，所以线性对数的假设对于所有的 n 都是合理的。

创新练习

以下练习旨在增加读者快速解决典型问题的经验。考虑使用二分查找法和归并排序算法，或者设计自己的分治算法。实现并测试你的算法。

4.2.20 中位数。向 StdStats（程序 2.2.4）添加一个方法 `median()`，该方法以线性时间计算 n 个整数数组的中值。提示：归约为排序。

4.2.21 众数。在 StdStats（程序 2.2.4）中添加方法 `mode()`，该方法以线性时间计算 n 个整数数组的众数（最频繁出现的值）。提示：归约为排序。

4.2.22 整数排序。编写一个线性运行时间的过滤器，从标准输入读取一个位于 0 到 99 之间的整数序列，并在标准输出中输出排序后的整数。例如，如果输入序列为

```
98 2 3 1 0 0 0 3 98 98 2 2 2 0 0 0 2
```

你的程序应该打印输出序列

```
0 0 0 0 0 0 1 2 2 2 2 2 3 3 98 98 98
```

4.2.23 向下和向上舍入。给定一个包含 n 个可比较项的有序数组，请编写函数 `floor()` 和 `ceiling()`，以对数时间返回不大于（或小于）参数项的最大（或最小）项的索引。

4.2.24 双调最大值。如果一个数组由递增关键字序列紧跟一个递减关键字序列组成，那么称之为双调数组。给定一个双调数组，设计一个对数运行时间算法，以查找最大关键字的索引。

4.2.25 双调数组中的查找。给定一个包含 n 个不同整数的双调数组，设计一个对数时间算法，用于确定一个给定的整数是否在数组中。

4.2.26 最接近数对。给定一个 n 个实数的数组，设计一个线性时间算法，以找出一对值最接近的数。

563 4.2.27 最远离数对。给定一个 n 个实数的数组，设计一个线性时间算法，以找到一对值差别最大的数。

4.2.28 两数和。给定一个包含 n 个整数的数组，设计一个线性时间算法，以确定它们中是否存在任何两个数的和为 0。

4.2.29 三数和。给定一个包含 n 个整数的数组，设计一个算法，以确定它们中是否有三个数的和为 0。你的程序的运行时间应该正比于 $n^2 \log n$ 。加分题：编写一个以二次型时间解决问题的程序。

- 4.2.30 多数派。在一个长度为 n 的数组中，如果一个元素出现的次数超过 $n/2$ 次，则称为多数派。给定一个字符串数组，设计一个线性时间算法来识别多数派（如果存在的话）。
- 4.2.31 最大的空白间隔。给定 Web 服务器上某个文件被请求的 n 个时间戳，确定该文件没有被请求的最大时间间隔。编写一个程序以线性时间解决这个问题。
- 4.2.32 无前缀编码。在数据压缩中，如果没有字符串是另一个字符串的前缀，那么这组字符串是无前缀的。例如，字符串集合 $\{01, 10, 0010, 1111\}$ 是无前缀的，但字符串集合 $\{01, 10, 0010, 1010\}$ 不是无前缀的，因为 10 是 1010 的前缀。请编写一个程序，从标准输入中读取一组字符串，并确定该字符串组是否为无前缀编码。
- 4.2.33 分区。请设计一个线性时间算法，以对已知最多包含两个不同值的 Comparable 对象数组进行排序。提示：使用两个指针，一个从左端开始向右移动，另一个从右端开始向左移动。保持左指针左侧的所有元素等于两个值中较小的一个，并且右指针右侧的所有元素等于两个值中较大的一个。
- 4.2.34 荷兰国旗问题。设计一个线性时间算法，对已知最多有三个不同值的 Comparable 对象数组进行排序。（Edsger Dijkstra 将此称为荷兰国旗问题，因为该结果是很像国旗中三个条纹那样的三个值。）
- 4.2.35 快速排序。编写一个递归程序，使用之前练习中描述的分治算法作为子例程，对 Comparable 对象进行排序：首先，选择一个随机元素 v 作为分区元素。接下来，将数组划分为包含小于 v 的所有元素的左子数组，由包含所有等于 v 的元素组成的中间子数组，以及包含大于 v 的所有元素的右子数组。最后，对左右子数组进行递归排序。
- 4.2.36 反转域名。编写一个程序，从标准输入中读取一系列域名，并按顺序输出其反向域名。例如，域名 `cs.princeton.edu` 的反向域名是 `edu.princeton.cs`。此计算适用于 Web 日志分析。为此，请创建一个实现 Comparable 接口的数据类型 Domain（使用反向域名顺序存储域名信息）。
- 4.2.37 数组中的局部最小值。给定一个包含 n 个实数的数组，设计一个对数时间算法，以查找一个局部最小值（一个索引 i ，使得 $a[i] < a[i-1]$ 和 $a[i] < a[i+1]$ ）。
- 4.2.38 离散分布。设计一个快速算法，反复生成离散分布的数字：给定一个和为 1 的非负实数组成的数组 a ，目标是以概率 $a[i]$ 返回索引 i 。构造一个累积和的数组 $sum[]$ ， $sum[i]$ 是 $a[]$ 的前 i 个元素的和。现在，生成一个 0 到 1 之间的随机实数 r ，并使用二分查找来返回满足 $sum[i] \leq r < sum[i+1]$ 的索引 i 。将这种方法的性能与 RandomSurfer（程序 1.6.2）中采用的方法进行比较。
- 4.2.39 隐含波动率。通常，波动率 σ 是布莱克-斯科尔斯公式中的未知值（参见练习 2.1.28）。编写一个程序，从命令行读取 s 、 x 、 r 、 t 和欧式看涨期权的当前价格等参数，并使用二分查找法来计算 σ 。
- 4.2.40 渗透阈值。写一个 Percolation（程序 2.4.1）的客户程序，使用二分查找法来估计渗透阈值。

564

565

4.3 栈和队列

在本节，我们将介绍两个相互关联的数据类型：栈和队列，它们都可以来处理任意大小的对象集合。栈和队列都是集合的一些具体实现。我们将集合中的对象称为数据项（item）。一个集合具有四个基本操作：创建一个集合、插入一个项、删除一个项，以及测试集合是否为空。

当我们将一个项插入一个集合中时，目的是明确的。但是当删除一个项时，应该删除哪一个呢？每种类型的集合的删除规则决定了它的特征，并且每种集合也有针对不同性能要求

的实现。在现实生活中，你已经遇到过很多种不同的集合和数据项的删除方式，只是你还没有意识到。

例如，队列的删除规则是总是删除集合中存在最长时间的项。这种策略称为“先进先出”（First In First Out，FIFO）。人们排队买票就是遵循这一原则：队列按到达顺序排列，因此离开队列的人比其他排队的时间长。

栈的删除规则则是一个完全不同的策略：总是删除进入集合时间最短的项。这个策略被称为“后进先出”（Last In First Out，LIFO）。例如，当进入和离开飞机客舱时遵循的策略接近于后进先出：靠近船舱前部的人登机最晚但离开最早。

栈和队列的应用十分广泛，因此，熟悉其基本属性以及每种适用的场景十分重要。它们都是可以用来解决更高级别编程任务的基本数据结构。它们被广泛应用于系统和应用程序设计，在本节和 4.5 节的几个例子中会涉及。

566

下推栈 下推栈（或者简称为栈）是一种基于后进先出（LIFO）策略的集合。

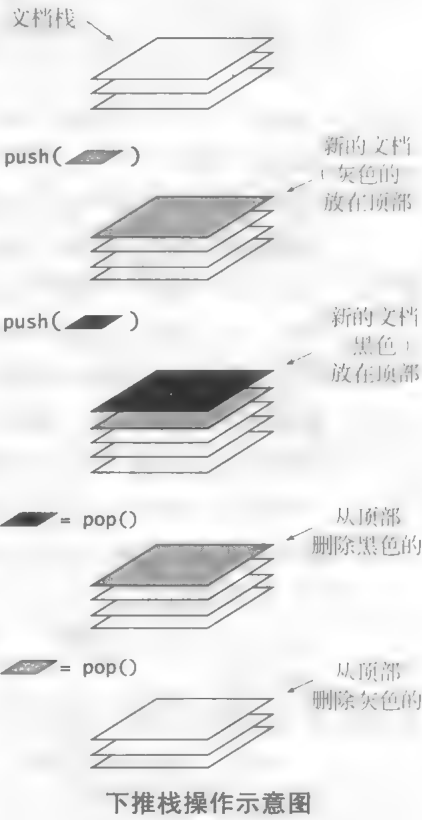
LIFO 策略是经常使用的几个计算机应用程序的基础。例如，许多人把电子邮件作为一个栈来组织，当接收到一个新邮件时，会把它放在最上面，位于最上面的邮件先被处理，即最近的日期优先（后进先出）。这个策略的优点是我们可以尽快看到新的邮件，缺点是如果不清空堆栈，一些旧的邮件可能永远不会被阅读。

浏览网页时，可能还会遇到另一种类型的栈的应用。当点击一个超链接时，浏览器会显示一个新页面（并将其插入一个栈中）。继续点击超链接可访问新页面，但可以随时通过点击后退按钮（从栈中删除）重新访问上一页。下推栈提供的后进先出策略正是你所期望的行为。

这些栈的使用方法非常直观，但也许不太有说服力。事实上，栈在计算中是一种重要而基础的数据结构，但我们在本节的后半部分再探讨它的应用。目前，我们的目标是理解栈的工作方式，以及学会如何实现栈。

栈从计算机出现的早期就被广泛使用。按照传统，我们把栈中插入项的操作命名为 `push`（又称入栈、压栈等），栈删除操作命名为 `pop`（又称出栈或弹出），栈数据类型 API 如下所示：

567



```
public class *StackOfStrings
{
    *StackOfStrings()    新建一个空栈
    boolean isEmpty()    栈是否为空
    void push(String item)  将一个字符串进栈
    String pop()          删除并返回最近插入的字符串
}
```

字符串下推栈的 API

星号表示我们将讨论这个 API 的多种实现（我们在本节中讨论三个：ArrayStackOfStrings、LinkedStackOfStrings 和 ResizingArrayStackOfStrings）。这个 API 中还包含了一个方法来测试堆栈是否为空，建议客户程序在调用 pop() 等方法之前使用 isEmpty() 来检查，以避免为空栈调用了这些操作方法。

这个 API 有一个严重的限制，使得在应用程序中并不方便使用：我们希望有一个包含多种数据类型的栈，而不仅仅是字符串。我们将在本节稍后描述如何消除这个限制（以及这么做的重要性）。

数组实现 用数组表示堆栈是一个很自然的想法，但在进一步阅读之前，我们有必要思考如何实现类 ArrayStackOfStrings。

我们遇到的第一个问题可能是如何实现构造函数 ArrayStackOfStrings()。显然，你需要一个实例变量、字符串数组 items[] 来保存栈的各个项，但该数组应该有多大？一种解决方案是先初始化一个长度为 0 的数组，并确保数组长度始终等于栈的大小，但该解决方案需要在每个 push() 和 pop() 操作时分配一个新的数组并将所有的项复制到其中。其实这样做非常低效和烦琐，而且也没有必要。为了简单解决这个问题，我们可以通过让客户程序为构造函数提供一个参数，用于表示栈的最大项个数。

我们的下一个问题是数据项的顺序。我们可能会源于自然的想法，将数组中的 n 个项按插入顺序保存，其中最近插入的项在 item[0] 位置，最早插入的项在 item[n-1] 位置。但是，每次调用 push 或 pop 时，都必须移动所有其他项以反映栈的新状态。更简单、更有效的方法是将项目按相反的顺序保存，即将最近插入的项保存在 item[n-1] 中，最早插入的保存在 item[0] 中。这个策略使得能够在数组的末尾添加和删除项，而不移动数组中的任何其他项。

568

几乎没有比 ArrayStackOfStrings（程序 4.3.1）中的栈更简单的实现了——所有方法都只用一行代码实现！实例变量是一个保存栈中项的数组 items[]，以及对栈中项的数量进行计数的整数 n 。要删除一个项目，我们递减 n ，然后返回 item[n]；如果要插入一个新的项目，我们设置 item[n] 等于新的项，然后增加 n 。上述操作还有以下属性：

- 堆栈中的项数为 n 。
- 当 n 为 0 时，栈为空。
- 栈中的项按插入顺序存储在数组中。
- 最近插入的项（如果堆栈是非空的）是 item[n-1]。

像往常一样，使用一个常量序列来推演这个过程，是验证我们的实现是否与预期相一致的一种最简单的方法。请你一定要完全理解这段代码实现的原理。为了帮助你理解，也许最好的方法就是分析一系列 push() 和 pop() 操作后栈内容的跟踪信息。ArrayStackOfStrings 的测试客户程序允许使用任意的操作序列进行测试：对标准输入中的每个字符串进行 push()（除了前面带减号的字符串），对减号字符为第一个字符的字符串则执行 pop() 操作。

StdIn	StdOut	n	items[]				
			0	1	2	3	4
		0					
to		1	to				
be		2	to	be			
or		3	to	be	or		
not		4	to	be	or	not	
to		5	to	be	or	not	to
-	to	4	to	be	or	not	to
be		5	to	be	or	not	be
-	be	4	to	be	or	not	be
-	not	3	to	be	or	not	be
that		4	to	be	or	that	be
-	that	3	to	be	or	that	be
-	or	2	to	be	or	that	be
-	be	1	to	be	or	that	be
is		2	to	is	or	not	to

这种实现的主要特点是 push 和 pop 操作 ArrayStackOfStrings 的测试客户程序的跟踪信息

运行时间是常量。缺点是它要求客户程序提前估计栈的最大大小，并且使用的空间总是与这个最大值成正比，这在某些情况下可能并不合理的。在 `push()` 的代码中，我们省略了检测栈是否已经存满的代码。在稍后给出的实现中，我们将用一种新的方法来弥补这个缺陷，设计一种不会被填满的栈（除非在极端的情况下 Java 没有可用的内存空间）。

569

程序4.3.1 字符串（数组）栈

```
public class ArrayStackOfStrings
{
    private String[] items;
    private int n = 0;

    public ArrayStackOfStrings(int capacity)
    { items = new String[capacity]; }

    public boolean isEmpty()
    { return (n == 0); }

    public void push(String item)
    { items[n++] = item; }

    public String pop()
    { return items[--n]; }

    public static void main(String[] args)
    { // 按照设定的容量创建一个栈，并按照标准输入的指令
      // 进行字符串的push或pop操作
      int cap = Integer.parseInt(args[0]);
      ArrayStackOfStrings stack = new ArrayStackOfStrings(cap);
      while (!StdIn.isEmpty())
      {
          String item = StdIn.readString();
          if (!item.equals("-"))
              stack.push(item);
          else
              StdOut.print(stack.pop() + " ");
      }
    }
}
```

items[]	栈的项
■	项数
items[n-1]	最近插入的项

如本代码所示，栈方法可以用单行代码实现。客户程序按照标准输入的指示调用push或pop字符串（减号表示pop，任何其他字符串表示push）。push()中省略了用来测试堆栈是否已满的代码（见正文）。

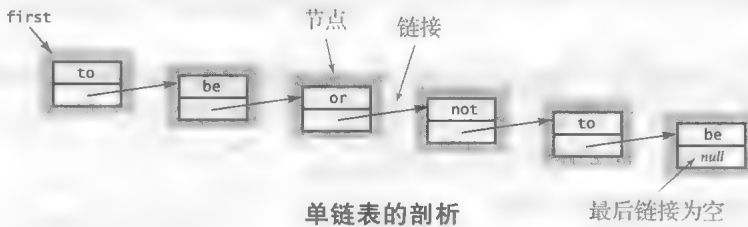
```
% more tobe.txt
to be or not to - be - - that - - - is
% java ArrayStackOfStrings 5 < tobe.txt
to be not that or be
```

570

链表 在设计集合（如栈和队列）时，一个重要的目标是确保所使用的内存量与集合中的项数成正比。ArrayStackOfStrings 中实现的栈使用固定长度的数组，因而难以实现这个目标，特别是当栈为空或几乎为空时，这种方式可能会浪费大量的内存。此属性使得通过固定长度数组实现栈难以适合许多应用程序。现在我们使用一个称为链表（linked list）的基本数据结构，基于它来完成集合（特别是栈和队列）的实现，从而实现本段开头所提出的目标。

一个单链表中包含了一个节点的序列，每个节点包含一个对其后继者的引用（或链接）。按照规定，最后一个节点的链接为 `null`，以表示链表终止。其中，节点是一个抽象实体，它包括了两部分，一部分是用于构建链表结构的链接，此外还可以用来保存任何类型的数据。当跟踪使用链表和其他链接结构的代码时，我们使用一种可视化的表示方法：

- 绘制一个矩形来表示每个链表节点。
- 把数据项和链接放在矩形内。
- 使用一个箭头指向被引用的对象来表示链接的引用关系。



这种可视化的表示方式表达了链表的基本特征，并将焦点放在链接上。例如，上图中展示了一个单链表，其中包含了 to、be、or、not、to 和 be 几个单词组成的序列。

使用面向对象程序设计实现链表并不困难。我们为本质上递归的节点抽象定义了一个类。与递归函数一样，递归数据结构的概念起初可能有点让人难以理解。

```
class Node
{
    String item;
    Node next;
}
```

一个 Node 对象有两个实例变量：一个 String 和一个 Node。String 实例变量是想用链表存储的任何数据的占位符（可以使用任何一组实例变量来替代）。Node 类型的实例变量 next 描述了数据结构的链接性质：它用于存储链表中后续节点的引用（或用 null 以表示没有后续节点）。通过使用这个递归定义，我们就可以用 Node 类型的变量来表示一个链接，其中这个变量的 next 只有两种情况，或者其值为 null，或者存储的是对下一个 Node 类型变量的引用，而这个变量的 next 域也可能指向其他链表。

571

为了强调只是使用 Node 类来表示数据结构，我们不为它定义任何实例方法。与任何类一样，我们可以通过 new Node() 调用（无参数）构造函数来创建 Node 类型的对象，这个函数返回一个新的 Node 对象，它的实例变量都被初始化为默认值 null。

例如，要建立一个包含 to、be 和 or 的链表，我们首先为每个项创建一个 Node：

```
Node first = new Node();
Node second = new Node();
Node third = new Node();
```

并将每个 Node 对象中的 item 实例变量赋予所需的值：

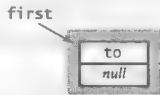
```
first.item = "to";
second.item = "be";
third.item = "or";
```

然后通过指定其实例变量 next 来构建链表：

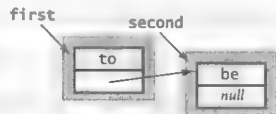
```
first.next = second;
second.next = third;
```

结果是，first 是对三节点链表中第一个节点的引用，

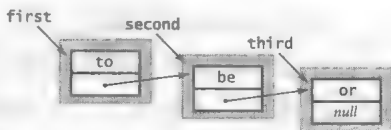
```
Node first = new Node();
first.item = "to";
```



```
Node second = new Node();
second.item = "be";
first.next = second;
```



```
Node third = new Node();
third.item = "or";
second.next = third;
```



链表各节点相链接的示意图

second 是对第二个节点的引用，third 是对最后一个节点的引用。下图的代码会执行相同的赋值语句；只是顺序不同。

572

链表可以用于表示一系列数据项。在上述例子中，first 表示 to、be 和 or 的序列。或者，我们也可以使用数组表示与上例相同的字符串序列。例如，我们可以使用

```
String[] items = { "to", "be", "or" };
```

表示相同的序列。这两者的不同之处在于，使用链表将项插入序列中及从序列中删除项更为容易。接下来，我们讨论实现这两个任务的代码。

假设我们需要向链表中插入一个新的节点。最简单的方式是在链表的头部进行操作。例如，要在第一个节点为 first 的给定链表的开始位置插入一个字符串 not，先将 first 节点保存在一个临时变量 oldFirst 中，然后创建一个新的 Node 对象，其 item 实例变量设置为 not，next 实例变量赋值为 oldFirst。

现在，假设要从链表中删除第一个节点。此操作更加简单：只需先将 first 赋值给 first.next。通常，在这个操作之前，一般先保存该节点的 item 值（把它赋值给某个变量），因为一旦更改了 first 变量，可能无法再访问该节点了。通常，该节点对象变成了孤立对象，它占用的内存最终会由 Java 内存管理系统回收。

573

在链表的头部插入和删除一个节点的代码仅仅涉及一些赋值语句，因此其操作时间为常量（与列表的长度无关）。如果有指向列表任意位置的节点引用，则可以使用同样的方法（其实会稍微复杂一些）在该节点之后移除或插入一个节点，且操作时间为常量。这些实现会被留作练习（参见练习 4.3.24 和练习 4.3.25），因为在链表头部插入和移除节点是实现栈所需的唯一操作。

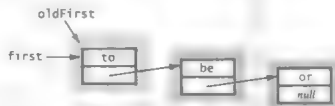
用链表实现栈。LinkedStackOfStrings（程序 4.3.2）使用链表来实现一个字符串栈，使用的代码比使用固定长度数组的基本解决方案略多。

实现的代码中使用了一个嵌套类 Node，等同于我们前面使用过的 Node 类。Java 允许以这种自然的方式，在类中定义和使用其他类。该类是私有的（private），因为客户程序不需要知道链表的任何细节。私有嵌套类的一个特点是，它的实例变量可以在定义的类中直接访问，但是不能在类以外的其他地方访问，因此不需要将 Node 实例变量显式地声明为 public 或 private（但这样做没有坏处）。

LinkedStackOfStrings 本身只包含一个实例变量：一个链表的引用，用于存储栈中的项。这个引用足以用来直接访问栈顶部的项，同时提供了访问栈中的其他项目所需要的 push() 和 pop()。同样，请你一定要完全理解这段代码实现的原理——这是使用链表结构的其他几种实现的原型（我们将在本章后面介绍这些实现方式的细节）。使用抽象的可视化列表表示来

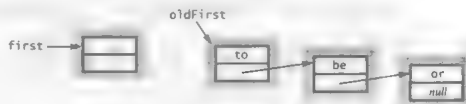
将链表中的第一个节点的引用保存下来

```
Node oldFirst = first;
```



在开始处创建一个新节点

```
first = new Node();
```



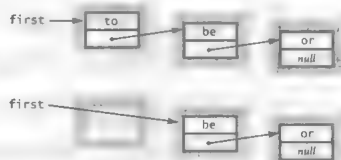
设计新节点中的实例变量

```
first.item = "not";
first.next = oldFirst;
```



在链表的开始处插入新节点

```
first = first.next;
```



删除链表中的第一个节点

追踪代码是分析和理解代码原理最好的方法。

链表遍历。我们在集合上执行的最常见操作之一就是遍历集合中的项目。例如，我们期望实现每个 Java API 中固有的 toString() 方法，以便通过跟踪来调试栈的代码。对于 ArrayStackOfStrings，这个实现非常简单。

574

程序4.3.2 字符串（链表）栈

```
public class LinkedStackOfStrings
{
    private Node first;
    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    { return (first == null); }

    public void push(String item)
    { // 在链表开始处插入新节点
      Node oldFirst = first;
      first = new Node();
      first.item = item;
      first.next = oldFirst;
    }

    public String pop()
    { // 从链表中移除第一个节点并返回项
      String item = first.item;
      first = first.next;
      return item;
    }

    public static void main(String[] args)
    {
        LinkedStackOfStrings stack = new LinkedStackOfStrings();
        // 测试客户程序见程序4.3.1
    }
}
```

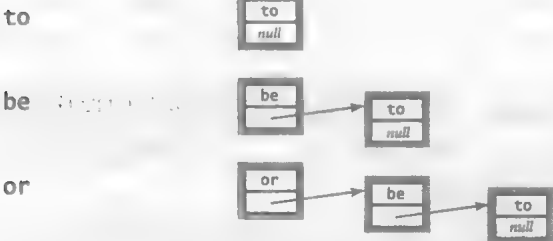
first	链表的第一个节点
item	链表项
next	链表的下一个节点

在此栈实现中，使用私有嵌套类Node作为基础实现了一个Node对象的链表，并用它来表示一个栈。实例变量first指向链表中的第一个（最近插入的）Node对象。每个Node对象中的实例变量next指向下一个Node节点（最后一个节点的next值为null）。不需要显式构造函数，因为Java默认将实例变量初始化为null

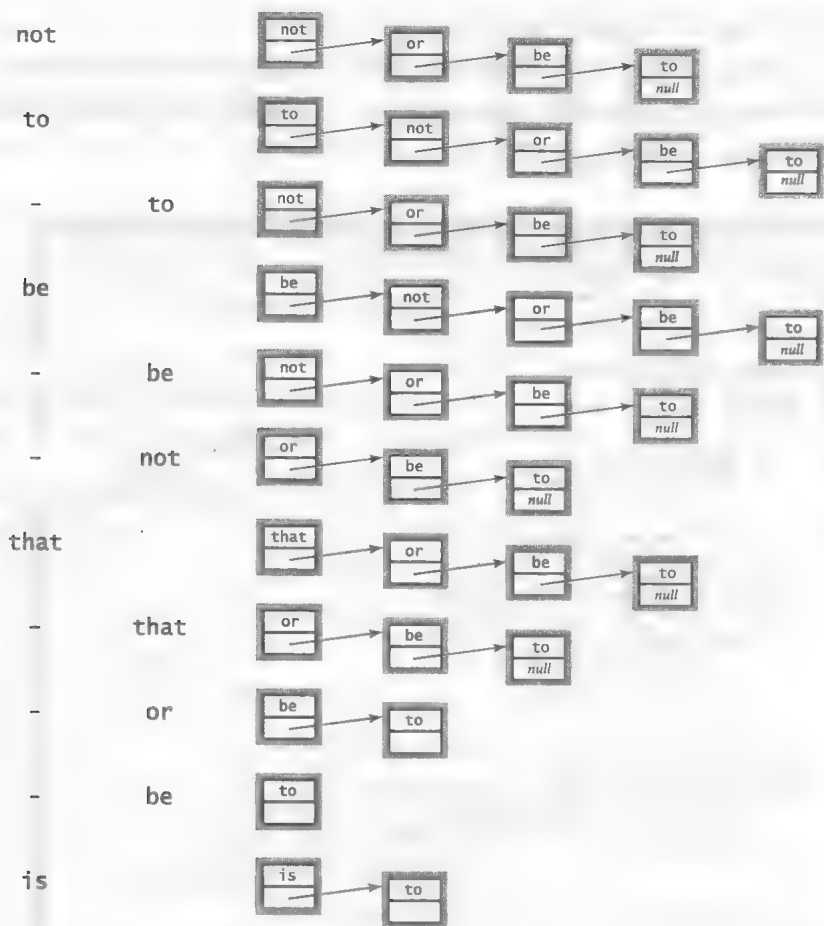
```
% java LinkedStackOfStrings < tobe.txt
to be not that or be
```

575

StdIn StdOut



LinkedStackOfStrings 测试客户程序的跟踪



576 LinkedStackOfStrings 测试客户程序的跟踪 (续)

上述解决方案仅适用于 n 较小的情况，因为每个字符串的连接都需要线性时间，因此总共需要二次量级时间。

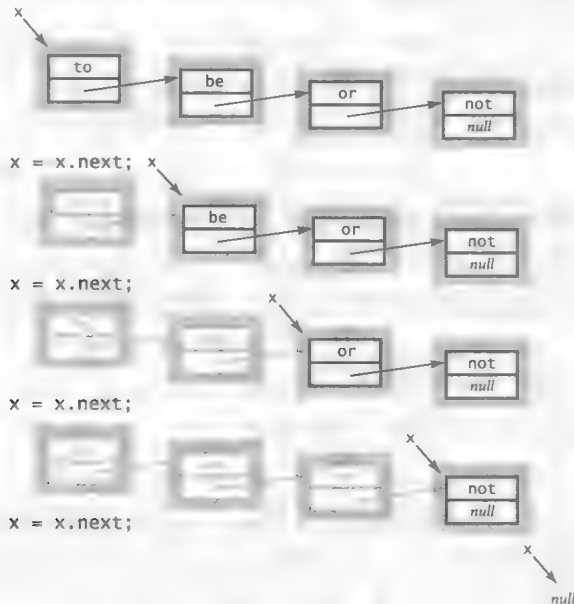
我们现在只关注于检查每个数据项的过程。访问链表中的项有一个相应的习惯用法：我们初始化一个循环索引变量 x ，让它指向链表的第一个 Node；然后，通过访问 $x.item$ 以获取 x 所指向的项的值，然后更新 x 来指向链表中的下一个 Node，即把 x 赋值为 $x.next$ ，重复这个过程，直到 x 为 null（这表明已经到达链表的末尾）。这个过程被称为链表遍历 (traversing)，在 `LinkedStackOfStrings` 的 `toString()` 实现中简洁地表达了这个过程：

```
public String toString()
{
    String s = "";
    for (int i = 0; i < n; i++)
        s += a[i] + " ";
    return s;
}
```

一旦你开始经常使用链表进行编程，这个习惯用法将变得像我们迭代访问数组元素的代码一样常用。在本节的最后，我们会讨论迭代器的概念，它允许我们编写客户程序代码来遍历集合中的项，而无须在这个细节上进行编程。

借助栈的链表实现，我们可以实现栈中使用大量数据项的客户程序，而不必担心内存的使用。同样的原则适用于各种类型的集合，因此链表被广泛应用于程序设计中。事实上，

Java 内存管理系统的典型实现就是维护了一个内存块的链表，其中的每一项对应于各种大小的内存块。在像 Java 这样的高级语言被广泛使用之前，内存管理的细节和链表的编程实现往往是任何程序员需要考虑的关键问题。在现代系统中，这些细节大多被封装在一些数据类型的实现中，比如本节的下推栈，以及后面会讲到的队列、符号表和集合等。如果你还学习了算法和数据结构课程，那你还会学到其他更多的数据类型，并且需要亲自动手创建和调试处理链表的程序。目前，你可以将注意力集中在理解链表在实现这些基本数据类型时的作用上。对于栈来说，链表非常重要，因为这允许在常量时间内实现 `push()` 和 `pop()` 方法，同时只使用一个小的常数因子的额外空间（用于存储 `next` 引用）。



链表的遍历

可变数组 接下来，我们考虑一种替代方法，以适应任意增长和收缩的数据结构，这是一个非常有吸引力的数据结构，是替代链表的一个很好的选择。同链表一样，我们现在介绍这种方法是因为在实现栈的过程中理解它的原理并不困难，而这种数据结构本身非常重要，在实现比栈更复杂的数据类型时能发挥更加重要的作用。

这个想法即修改基于数组的实现（程序 4.3.1），动态地调整数组 `items[]` 的长度，使其可以足够大，从而容纳所有的数据项；也可以在项数较少时压缩长度，不会浪费过多的内存。在 `ResizingArrayStackOfStrings`（程序 4.3.3）的代码中可以看到，实现这些目标并不困难。

首先，在 `push()` 中检查数组是否太小。具体来说，我们通过检查栈大小 `n` 是否等于数组长度 `items.length`，以检查数组中是否有空间容纳新数据项。如果有，则用代码 `items[n++] = item` 将新项插入；如果没有，我们创建一个长度是原数组两倍的新数组，并将原来栈中的项复制到新数组中，重置 `items[]` 实例变量使其引用新数组，从而使数组的长度加倍。

类似地，在 `pop()` 中，我们首先检查数组是否太大，如果是，就将其长度减半。简单考虑一下，你就会想到这里应该做的检查是栈的大小是否小于数组长度的 $1/4$ 。然后，在数组减半之后其状态大约半满，因此可以容量一定量的 `push()` 和 `pop()` 操作，而不必频繁改变数组的长度。这个特性非常重要。例如，如果使用的策略是在栈大小为数组长度一半时将数组减半，那么生成的数组会是满的，这意味着如果下一个操作是 `push()`，就会立即需要将数组长度加倍，那么这样的机制就导致系统陷入一个开销巨大的加倍和减半的循环中。

577
578

程序4.3.3 字符串（可变数组）栈

```
public class ResizingArrayStackOfStrings
{
    private String[] items = new String[1];
    private int n = 0;

    public boolean isEmpty()
    { return (n == 0); }

    private void resize(int capacity)
    { // 把栈中元素移到一个指定容量的新数组中
      String[] temp = new String[capacity];
      for (int i = 0; i < n; i++)
        temp[i] = items[i];
      items = temp;
    }

    public void push(String item)
    { // 将数据项插入栈中
      if (n == items.length) resize(2*items.length);
      items[n++] = item;
    }

    public String pop()
    { // 将最近插入的元素删除并返回
      String item = items[--n];
      items[n] = null; // 避免数据对象“游离”（见正文）
      if (n > 0 && n == items.length/4) resize(items.length/2);
      return item;
    }

    public static void main(String[] args)
    {
        // 见程序4.3.1的测试客户程序
    }
}
```

items[] | 栈中项
n | 栈中项数

这种实现的目标是支持任何大小的栈,而且不会过度浪费内存。它在栈被填满时将数组的长度增加一倍,并在数据项数减少时将数组的长度减半以使其始终保持至少四分之一被填满。平均而言,所有操作都只需要常量时间（请参阅正文）

% java ResizingArrayStackOfStrings < tobe.txt
to be not that or be

StdIn	StdOut	n	items. length	items[]							
				0	1	2	3	4	5	6	7
		0	1	null							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	null				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2	2	to	is						

ResizingArrayStackOfStrings 的测试客户程序的跟踪信息

摊销分析。这种加倍和减半的策略是在浪费空间（把数组的长度设置得太大而留下空位）和浪费时间（在每次插入之后重新组织数组）之间的明智折中。ResizingArrayStackOfStrings 中的特定策略保证栈永远不会溢出且永远不会少于四分之一满（除非栈为空，在这种情况下数组长度为 1）。如果你爱好数学，可能会喜欢用数学归纳来证明这个结论（见练习 4.3.18）。更重要的是，我们可以证明，加倍和减半的成本总是能够被其他栈操作的成本所吸收（在一个常数因子内）。我们把证明的细节留作数学上的练习，但这个想法很简单：当 push() 将数组的长度加倍到 n 时，它会从堆栈中的 $n/2$ 项开始，所以客户程序至少需要进行 $n/2$ 次额外的 push() 调用（如果中间调用了一些 pop() 则更多），数组的长度才能再次加倍。如果我们将数组加倍产生的开销平均到这 $n/2$ 个 push() 操作上，那么显然会得到一个常数。换句话说，在 ResizingArrayStackOfStrings 中，所有栈操作的总开销除以操作次数，得到的平均开销的上限会是一个常数。这个结论并不像说每个操作都需要一定的时间那样肯定，但其在许多应用程序（例如，当我们主要关注应用程序的总运行时间时）中具有相同的含义。这种分析被称为摊销分析——可变数组数据结构是很有价值的典型例子。

579
580

孤立项。Java 的垃圾收集策略是回收不能被访问的对象所占据的内存。在一开始我们实现 ArrayStackOfStrings 的 pop() 时，对弹出项的引用仍保留在数组中。该项是一个孤立项——我们将永远不会在类中再次使用它，要么是因为栈将会收缩，要么因为栈将被另一个引用覆盖（如果栈增长）——但 Java 的垃圾回收器无法知晓这一点。即使客户程序已经完成了对该项的所有操作，数组中的引用也可能使其保持活跃状态。这种情况（保留对不再需要的数据项的引用）称为“游离”，这与内存泄漏不同（即使内存管理系统没有对该项的引用）。在这种情况下，游离是很容易避免的。在 ResizingArrayStackOfStrings 的 pop() 实现中，将已弹出项对应的数组元素设置为 null，从而覆盖未使用的引用，使系统能够在客户程序完成时回收与弹出项关联的内存。

借助栈的可变数组实现（与链表实现时一样），我们可以编写使用栈的客户程序，而无需担心内存使用情况。同样的原则也适用于任何种类的集合。对于一些比栈更复杂的数据类型，可变数组优于链表，因为能够在常量时间内（通过索引）访问数组中的任何元素，这对于实现某些操作非常重要（例如，参见练习 4.3.37 的 RandomQueue）。与链表一样，最好将可变数组代码保存为基本数据类型，当需要时你就可以在客户代码中直接使用它们。

581

参数化的数据类型 我们已经开发并实现了一个基于特定类型（字符串）的栈。但是在开发客户程序时，常常需要实现其他类型数据的集合，而不一定是字符串。商业交易处理系统可能需要维护客户、账户、商家和交易的集合；大学课程安排系统可能需要维护班级、学生和房间的集合；便携式音乐播放器（mp3）可能需要维护歌曲、艺术家和专辑的集合；一个科学数据分析程序可能需要保存 double 值或 int 值的集合。在你遇到的编程任务中，你可能会需要维护任何类型的数据类型构成的集合，甚至是你自己创建的类型。你应该怎么做？在讨论了使用 Java 语言构造的两个简单方法（以及它们的缺点）之后，我们将引入一个更高级的数据结构，可以帮助我们正确地解决这个问题。

为每个项数据类型创建新的集合数据类型。我们可以创建类 StackOfInts、StackOfCustomers、StackOfStudents 等，当作 StackOfStrings 的补充。这种方法要求我们复制每种数据类型的代码，这违反了软件工程的基本设计准则，我们应该尽可能复用（而不是复制）代码。对于要放在栈上的每种类型的数据，我们都需要一个不同的类，因此，维护代码就变成了一个噩梦：每当你想要或需要进行更改时，必须在代码的每个版本中都这样做。尽管如

此，这种方法仍然被广泛使用，因为许多编程语言（包括 Java 的早期版本）没有提供任何更好的方法来解决这个问题。打破这个障碍成为一个程序员和一个编程环境成熟的标志。我们是否可以只用一个类来实现一系列字符串、整数以及任何类型的栈？

使用对象集合。我们可以开发一个栈，其数据项都是 `Object` 类型。使用继承，我们可以合法地压入任何类型的对象（如果我们要压栈一个 `Apple` 数据类型的对象，这是一个合法操作，因为 `Apple` 是 `Object` 的子类，其他类也是如此）。当数据项弹出栈时，我们必须将其转换回正确的类型（堆栈中的所有项都是 `Object` 类型，但是代码处理的是 `Apple` 类型的对象）。总之，如果我们能将 `*StackOfStrings` 实现中所有 `String` 更改为 `Object`，就能创建一个

[582] `StackOfObjects` 类，我们就可以编写以下类似的代码来使用它：

```
StackOfObjects stack = new StackOfObjects();
Apple a = new Apple();
stack.push(a);
...
a = (Apple) (stack.pop());
```

这样做可以实现我们的目标，即使用一个类就能实现对任何类型的对象构成的栈的创建和操作。但这种方法是不可取的，因为它将会给客户程序带来编码错误的风险，而这些错误无法在编译时检测到。例如，我们无法阻止程序员将不同类型的对象放在同一个栈上，如下例所示：

```
ObjectStack stack = new ObjectStack();
Apple a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b);
a = (Apple) (stack.pop()); // 抛出异常 ClassCastException
b = (Orange) (stack.pop());
```

以这种方式进行类型转换等于假设客户程序将从栈中弹出的对象强制转换为正确的类型，因此关闭了 Java 类型系统提供的保护。程序员使用类型系统的一个原因是防止由这种隐式假设而引起的错误。如果在编译时无法对代码进行类型检查，那么在某些代码中可能会出现错误的强制转换，这些代码可能在很多情况下都可以正常运行，仅在某些特定的运行环境时才会出错。我们要尽量避免这样的错误，因为这些错误可能在程序交付给客户之后很久才出现，而客户完全无法解决。

Java 泛型。Java 中提供了一个名为“泛型”类型的特定机制来解决我们所面临的问题。有了泛型，我们就可以按照客户代码指定的类型构建对象集合。这样做的主要好处是能够在编译时（软件开发时）发现类型不匹配错误，而不是在运行时（当客户程序正在使用该软件时）发现类型不匹配错误。从概念上讲，泛型有点令人困惑（它们对编程语言有足够深刻的影响，甚至在 Java 的早期版本中都没有提供支持），但我们在当前上下文中使用它们时，仅引入了少量的额外内容，使用的 Java 语法也很容易理解。我们将泛型类的栈命名为 `Stack`，并为栈中的对象类型选择通用名称 `Item`（可以使用任何名称）。`Stack`（程序 4.3.4）的代码与 `LinkedStackOfStrings` 的代码基本相同（我们删去了 `Linked` 修饰符，因为我们的实现非常完善，客户程序不必关心数据的存储方式），只是我们将每个出现的 `String` 替换为 `Item`，并使用以下第一行代码声明该类：

```
public class Stack<Item>
```

程序4.3.4 泛型栈

```

public class Stack<Item>
{
    private Node first;

    private class Node
    {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    { return (first == null); }

    public void push(Item item)
    { // 将数据项插入栈中
      Node oldFirst = first;
      first = new Node();
      first.item = item;
      first.next = oldFirst;
    }

    public Item pop()
    { // 将最近插入的元素删除并返回
      Item item = first.item;
      first = first.next;
      return item;
    }

    public static void main(String[] args)
    {
        Stack<String> stack = new Stack<String>();
        // 见程序4.3.1的测试客户程序
    }
}

```

first | 链表上第一个节点

item	栈项
next	链表上的下一个节点

这个代码和程序4.3.2几乎是一样的，但是值得重复，因为它展示了使用泛型来允许客户程序支持任何类型的数据是多么容易。此代码中的关键字Item是一个类型参数，它是客户程序提供的实际类型名称的占位符。

```

% java Stack < tobe.txt
to be not that or be

```

Item 是一个类型参数，是客户程序指定的某些实际类型的占位符。可以将 Stack<Item> 读作“Item 的栈”，这正是我们想要的。在实现 Stack 时，我们不知道 Item 的实际类型，但是客户程序可以使用栈来处理任何类型的数据，包括在我们开发完 Stack 很长时间后才定义的数据类型。下面的客户代码中，在创建栈时指定了 Apple 类型参数：

```

Stack<Apple> stack = new Stack<Apple>();
Apple a = new Apple();
...
stack.push(a);

```

如果你尝试在堆栈上推送一个错误类型的对象，像这样：

```

Stack<Apple> stack = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b); // 编译时错误

```

你会得到一个编译时错误：

```
push(Apple) in Stack<Apple> cannot be applied to (Orange)
```

此外，在我们的 Stack 实现中，Java 可以使用类型参数 Item 来检查类型不匹配错误——例如，即使尚未知道实际类型，也必须为 Item 类型的变量分配 Item 类型的值。

自动装箱。像程序 4.3.4 这样的泛型代码有一个小麻烦，即类型参数必须是一个引用类型，那么我们怎样才能在代码中使用基本类型，如 int 和 double？Java 语言中提供了自动装箱（autoboxing）和自动拆箱（unboxing）机制，使我们能够在泛型代码中使用基本类型。Java 为每一个基本类型内置了一个包装类型（wrapper type），其对应于 boolean、byte、char、double、float、int、long 和 short 这些基本类型，分别是 Boolean、Byte、Character、Double、Float、Integer、Long 和 Short。Java 会自动在这些引用类型和相应的基本类型（在赋值语句、方法参数和算术 / 逻辑表达式中）之间进行转换，以便我们可以编写如下代码：

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);      // 自动装箱 (int -> Integer)
int a = stack.pop(); // 自动拆箱 (Integer -> int)
```

在这个例子中，当我们将原始值 17 传递给 push() 方法时，Java 自动将其转换（自动装箱）为 Integer 类型。pop() 方法返回一个 Integer，在将其赋值给变量 a 之前，Java 将其转换为 int 值。此功能便于编写代码，但是涉及大量的幕后处理，会影响性能。在一些性能关键的应用程序中，像 StackOfInts 这样的类还是很有必要的。

泛型提供了我们需要的解决方案：它们可以实现代码复用，同时提供类型安全性。仔细研究 Stack（程序 4.3.4），并确保你理解每一行代码，这会在将来的编程工作中给你很多帮助，因为参数化数据类型的使用是一个非常重要的高级语言编程技巧，而 Java 中也提供了非常完善的支持。你不必成为专家就可以利用这一强大的功能。

栈的应用 下推栈在计算中扮演着重要的角色。如果你学习了操作系统、编程语言或者其他计算机科学中的高级课题，你将会发现，在许多应用程序中，栈不仅是直接使用的，而且还被用作许多高级语言程序的执行基础，包括 Java 和 Python 等。

算术表达式。我们在第 1 章讨论过的一些初级程序中包含了算术表达式的求值计算，例如：

```
( 1+( ( 2+3 ) * ( 4 * 5 ) ) )
```

按照数学知识，用 4 乘以 5、3 加上 2，两个结果相乘，再加上 1，得到结果 101。但是 Java 如何进行计算呢？如果不深入了解 Java 的构建原理，我们可以编写一个 Java 程序，将字符串（表达式）作为 Java 程序的输入，输出是表达式产生的值，并借此来讨论其基本思想。为了简单起见，我们从下面的显式递归定义开始：一个算术表达式要么是一个数值，要么是一个左括号后跟一个算术表达式，后跟一个运算符，后跟另一个算术表达式，再后跟右括号。为了简单起见，这个定义适用于全括号（fully parenthesized）算术表达式，它借助括号精确地指定了哪些运算符适用于哪些操作数（即每一个运算，无论优先级，都用一对括号括起来——译者注）——我们对 1+2*3 这样的表达式更熟悉，在这样的表达式中我们利用了运算符优先级规则来描述计算的顺序，而不是括号。我们讨论的基本机制可以处理优先级，但为了简单起见，我们没有考虑这些情况。为了简洁性，我们仅支持常用的二元运算符 *、+ 和 -，以及只有一个参数的平方根运算符 sqrt。我们可以很容易地扩展到更多的运算符，以涵盖大量常用的数学表达式，包括除法、三角函数和指数函数等。我们的重点是理解如何

解释括号、运算符和数值的字符串，以便能够按照正确的优先级顺序执行任何计算机上都支持的低级算术运算。

算术表达式求值。我们如何才能将算术表达式（字符串）转换为它所代表的值？狄克斯特拉（Edsgar Dijkstra）在 20 世纪 60 年代开发了一种非常简单的算法，该算法使用了两个下推栈（一个用于操作数，另一个用于运算符）来完成这项工作。一个表达式由括号、运算符和操作数（数字）组成。该算法从左向右处理字符串，每次处理一个实体，我们根据四种可能的情况操作栈，如下：

- 将操作数压入操作数栈上。
- 将运算符压入运算符栈上。
- 忽略左括号。
- 在遇到右括号时，弹出一个运算符，弹出所需数量的操作数，用运算符和操作数完成相应的计算，并把计算结果压入操作数栈。

程序4.3.5 表达式求值

```
public class Evaluate
{
    public static void main(String[] args)
    {
        Stack<String> ops = new Stack<String>();
        Stack<Double> values = new Stack<Double>();
        while (!StdIn.isEmpty())
        {
            // 读入一个短串，若是运算符，则入栈
            String token = StdIn.readString();
            if (token.equals("("))
            else if (token.equals("+")) ops.push(token);
            else if (token.equals("-")) ops.push(token);
            else if (token.equals("*")) ops.push(token);
            else if (token.equals("sqrt")) ops.push(token);
            else if (token.equals(")"))
            {
                // 如果读到的是")", 则弹出，求值，并将结果入栈
                String op = ops.pop();
                double v = values.pop();
                if (op.equals("+")) v = values.pop() + v;
                else if (op.equals("-")) v = values.pop() - v;
                else if (op.equals("*")) v = values.pop() * v;
                else if (op.equals("sqrt")) v = Math.sqrt(v);
                values.push(v);
            }
            // 如果读到的不是运算符，也不是括号，那么按照double类型的数据入栈
            else values.push(Double.parseDouble(token));
        }
        StdOut.println(values.pop());
    }
}
```

ops	运算符栈
values	操作数栈
token	当前标记
v	当前值

这个Stack的客户程序从标准输入中读取一个全括号数值表达式，使用Dijkstra的双栈算法来求值，并将结果数字打印到标准输出。程序演示了一个基本的计算过程：将字符串解释为程序，并执行该程序以计算出所需的结果。执行一个Java程序只不过是这个过程的一个更复杂的版本。

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

```
% java Evaluate
( ( 1 + sqrt ( 5.0 ) ) * 0.5 )
1.618033988749895
```

当最后一个右括号处理完毕，栈中只剩下一个值，即表达式的值。Dijkstra 的双栈算法

	$(1 + ((2 + 3) * (4 * 5)))$
1	$+ ((2 + 3) * (4 * 5))$
1	$((2 + 3) * (4 * 5))$
+	
2	$+ 3) * (4 * 5))$
2	
-	$3) * (4 * 5))$
+	
1	$) * (4 * 5))$
3	
1	$* (4 * 5))$
5	
*	$(4 * 5))$
*	
4	$* 5))$
*	
5	$))$
5	
20	$))$
20	
100	$)$
100	
101	
101	

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
( 1 + ( 5 * ( 4 * 5 ) ) )
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

程序 Evaluate(程序 4.3.5) 是这个算法的一个实现。此代码是一个简单的解释器 (interpreter) 示例：一个程序执行另一个程序 (在本例中是一个算术表达式)，一次执行一步或一行。编译器是一个将程序从高级语言转换为可以完成这个工作的低级语言的程序。编译器的转换工作是一个比解释器的逐步转换更为复杂的过程，但两者基于相同的基础机制。Java 编译器将 Java 程序设计语言代码翻译成 Java 字节码。最初，Java 基于解释器进行工作。但是现在，Java 包含一个编译器，用于把算术表达式 (更一般地说，Java 程序) 转换为 Java 虚拟机 (这是一个在真实计算机上易于模拟的虚拟机器) 的低级代码。

$$(1((23+)(45*)*)+)$$

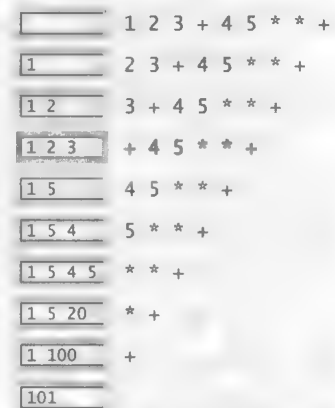
1 2 3 + 4 5 * * +

- 将操作数压入栈。
- 在遇到运算符时，弹出必要数量的操作数，并将运算符和这些操作数完成计算后的结果压入栈。

同样，这个过程最终在栈上只剩下一个值，即表达式的值。这种表示非常简单，以至于某些编程语言如 Forth（一种科学编程语言）和 PostScript（大多数打印机上使用的页面描述

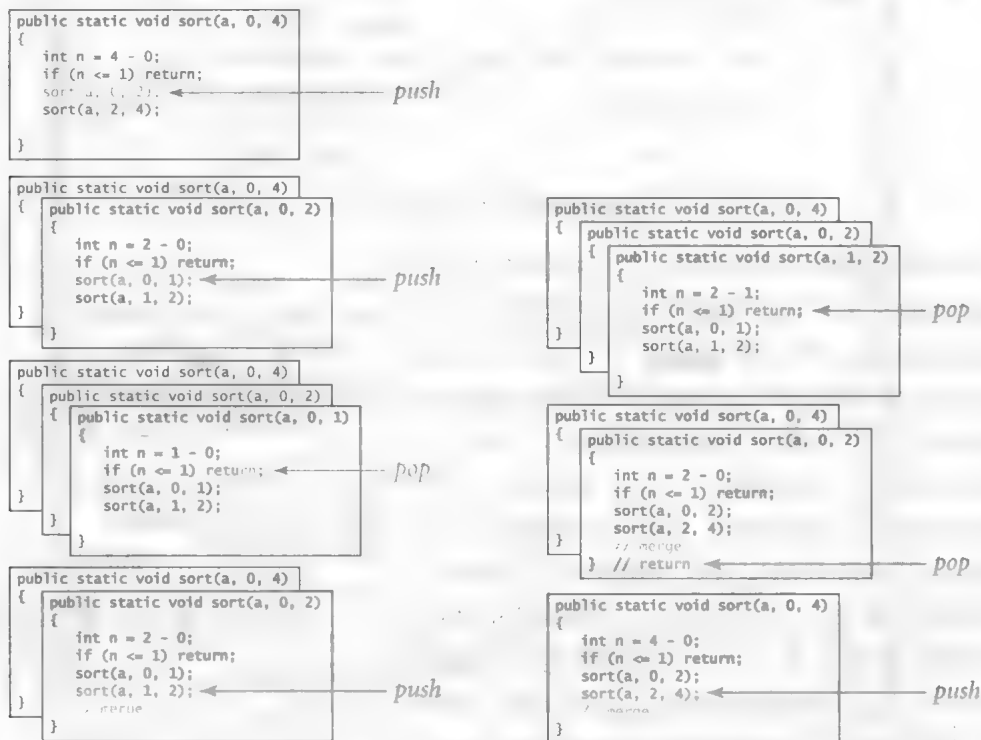
语言)使用显式栈作为主要的程序控制流管理结构。例如,字符串“1 2 3+4 5 * * +”是Forth 和 PostScript 中的合法程序,会得出值 101 并保存在执行栈上。在一些基于栈的编程语言中也采用类似的实现方式,因为它们对于许多类型的计算来说更简单。实际上,Java 虚拟机本身就是基于栈的方式实现的。

函数调用抽象。大多数程序隐式地使用栈,因为栈是用来实现函数调用的自然方式。在函数执行的过程中,我们定义其状态为所有变量的值和指向下一条执行指令的指针。计算环境的一个基本特征是每个计算完全由其状态(及其输入的值)决定。因此,系统可以通过保存状态来临时中止一个计算,然后通过恢复状态来恢复这个计算。如果你学习了操作系统课程,将会了解此过程的更多细节,这一技术是实现很多常用功能的关键所在(例如,从一个应用程序切换到另一个应用程序只是一个保存状态和恢复状态的问题而已)。使用栈来实现函数调用的抽象是一个很自然的方法。要调用一个函数,将调用时的状态压入栈。若要从函数调用中返回,则从栈中弹出状态,以便恢复函数调用之前的所有变量值,在包含函数调用的表达式中(如果有的话),还须将函数返回值(如果存在的话)替换入表达式中,然后继续执行下一条要执行的指令(其位置是计算保存状态的一部分)。只要发生了函数调用,这种机制即可递归地工作。事实上,如果你仔细思考这个过程,会发现这个过程和我们刚才详细讨论的表达式计算过程基本相同。一个程序就是一个复杂的表达式。



590

后缀表达式值的计算过程跟踪



使用栈实现函数调用

下推栈是一个基本的计算抽象。栈已成功应用于表达式计算、实现函数调用抽象,以及计算早期就存在的许多基本任务。我们将在 4.4 节讨论它的另一个重要应用(树遍历)。栈在

计算机科学的许多领域都有显式和广泛的应用，包括算法设计、操作系统、编译器，以及许多其他的计算应用。

591

FIFO 队列 FIFO 队列（也被简称为队列）是一个基于先进先出策略的集合。

按照到达先后顺序执行任务是日常生活中经常使用到的策略，从剧院中的排队人群到收费站排队的汽车，到你的计算机中等待应用程序处理的任务，都在使用“先来先服务”的原则。

任何服务政策的一个基本原则就是公平。大多数人想到公平的时候首先想到的是等候最长时间的人应该优先得到服务。这正是先入先出的原则，所以队列在许多应用程序中占据十分重要的地位。队列是许多日常现象的自然模型，在计算机问世之前，其特性已经得到深入的研究。

像往常一样，我们从描述一个 API 开始。依据传统，队列插入操作命名为入队（enqueue），删除操作命名为出队（dequeue），如以下 API 所示：

```
public class Queue<Item>
```

Queue()	创建一个空队列
boolean isEmpty()	队列是空的吗？
void enqueue(Item item)	将一个项插入队列中
Item dequeue()	移除并返回队列中最早加入的项
int size()	队列中项的个数

592

通用 FIFO 队列的 API

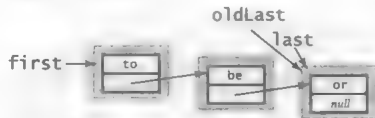
从 API 可以看到，我们将在实现中使用泛型，这样就可以编写客户程序来安全地创建和使用任何引用类型的队列。我们还增加了一个 size() 方法，而栈并没有这样的方法，因为队列客户程序通常要知道队列中项的数量，而大多数栈的客户程序并不需要（参见程序 4.3.8 和练习 4.3.11）。

根据从栈中获得的知识，我们可以使用链表或可变数组来开发队列的实现，其操作时间为常量，队列所占的内存大小随着队列中项的数量的增加或减少而变化。与栈一样，每一个实现都是一个经典的编程练习。在进一步阅读之前，你可以想一想如何实现这些代码。

链表实现。为了用一个链表实现一个队列，我们将这些项目按其到达先后顺序保存（与我们在 Stack 中使用的顺序相反）。dequeue() 的实现与 Stack

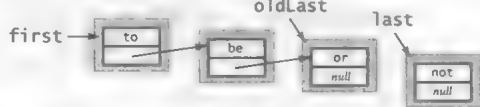


保存一个指向链表最后一个节点的链接
Node oldLast = last;

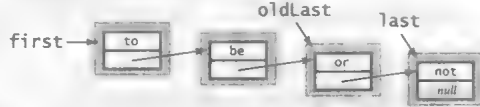


为最后一个节点创建一个新的节点

```
Node last = new Node();  
last.item = "not";
```



将新节点链接到链表的最后一个节点
oldLast.next = last;



在链表的末尾插入新节点

中的 `pop()` 实现相同（保存第一个节点的数据项，从队列中移除第一个节点并返回保存的项）。然而，实现 `enqueue()` 有点复杂：如何将一个节点添加到链表的末尾？为此，我们需要一个指向链表最后一个节点的链接，因为原来指向最后一个链接的链接将用来指向待插入的新节点。在 `Stack` 中，唯一的实例变量即指向链表第一个节点的引用；借助这些信息找到链表的末尾，唯一的办法就是遍历链表的所有节点直到最后。对于长链表，该解决方案效率低下。一个合理的替代方案是增加一个实例变量，用于指向链表的最后一个节点。额外增加一个需要维护的实例变量是不能掉以轻心的，特别是在链表代码中，因为每个修改链表的方法都需要通过代码来检查该变量是否需要修改（并进行必要的修改）。例如，移除链表的第一个节点可能涉及对最后一个节点引用的修改，因为如果链表只有一个节点，所以该节点既是第一个，也是最后一个（类似的细节使得链表代码难于调试）。`Queue`（程序 4.3.6）是 FIFO 队列 API 的链表实现，它具有与栈相同的性能属性：所有方法都是常量时间，内存使用量与队列大小成正比。

593

程序4.3.6 通用FIFO队列（链表）

```
public class Queue<Item>
{
    private Node first;
    private Node last;

    private class Node
    {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    { return (first == null); }

    public void enqueue(Item item)
    { // 在链表的末尾插入一个新节点
      Node oldLast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else oldLast.next = last;
    }

    public Item dequeue()
    { // 删除链表的第一个节点，并返回它的数据项
      Item item = first.item;
      first = first.next;
      if (isEmpty()) last = null;
      return item;
    }

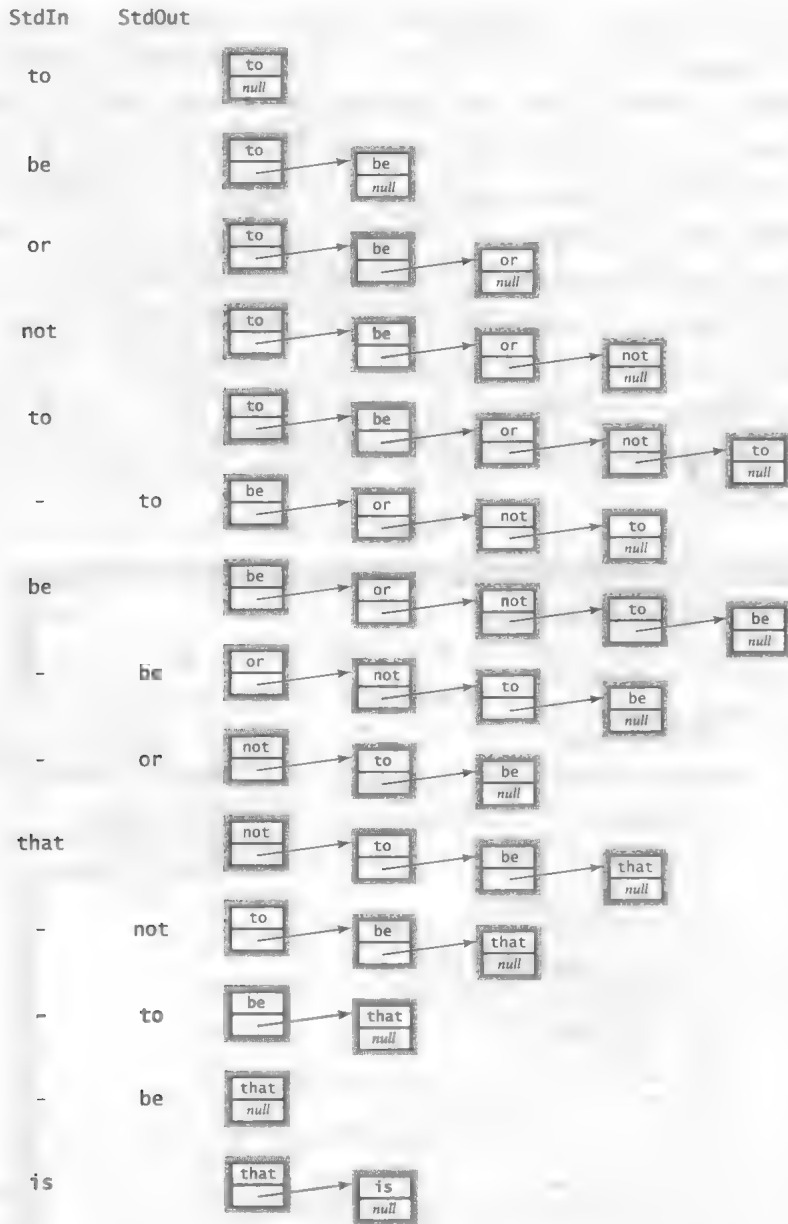
    public static void main(String[] args)
    { // 测试客户程序与程序4.3.2类似
      Queue<String> queue = new Queue<String>();
    }
}
```

first	链表中第一个节点
last	链表中最后一个节点
item	队列项
next	链表中下一个节点

此实现非常类似于栈的链表实现（程序 4.3.2）：`dequeue()`与`pop()`几乎相同，只是`enqueue()`会将新节点链接到链表的末尾，而不是如`push()`插入开头。为此，程序维护一个指向链表最后一个节点的实例变量`last`。`size()`方法留作练习（参见练习 4.3.11）

```
% java Queue < tobe.txt
to be or not to be
```

594



595

Queue 的测试客户程序的跟踪 (参见程序 4.3.6)

数组实现。我们其实还可以使用数组开发 FIFO 队列的实现方法，就像我们在 `ArrayStackOfStrings` (程序 4.3.1) 和 `ResizingArrayStackOfStrings` (程序 4.3.3) 中实现的不同的栈一样，它们有着相同的性能特征。这些实现可以留作编程练习，值得你进一步研究 (参见习题 4.3.19)。

随机队列。尽管 FIFO 和 LIFO 应用广泛，但它们的原理却并不神秘。考虑删除数据项时使用其他规则也是非常有意义的。考虑这样一种数据类型，方法 `dequeue()` 移除并返回一个随机项 (不重置抽样)，方法 `sample()` 返回一个随机项而不从队列中移除它 (重置抽样)。我们使用名称 `RandomQueue` 来表示此数据类型 (请参见练习 4.3.37)，这也是一种重要的数据结构。

栈、队列和随机队列的 API 本质上是相同的——它们仅在类和方法名称的选择上有所不同。这些数据类型之间的真正区别在于删除操作的语义——哪个项目将被删除？栈和队列

之间的区别从它们的名字和自然语言描述当中就能找到。这些区别类似于 $\text{Math.sin}(x)$ 和 $\text{Math.log}(x)$ 之间的差异，但是我们也也许希望使用栈和队列的形式化描述来阐明其含义（就像使用正弦和对数函数的数学描述一样的方法）。但是，精确地描述先入先出、后进先出或者随机移除并不是那么简单。对于初学者，你将使用哪种语言进行描述？英语？Java 或数学逻辑？描述程序如何运行的问题被称为规范化问题（specification problem），它直接引发了计算机科学中的深层问题。我们强调清晰和简洁代码的重要性的一个原因是，代码本身可以用作简单数据类型（如栈、队列和随机队列）的规范。

596

队列的应用 在过去的一个世纪，FIFO 队列被证明是在各种各样的应用中准确和有效的模型，从制造过程到电话网络到交通模拟，并且由此产生了一个被称为排队论的数学领域，它成功地用于理解和控制各种复杂的系统。FIFO 队列在计算中同样起着重要的作用。使用计算机时经常遇到队列：播放列表中的歌曲、要打印的文档或游戏中的事件。

也许队列最大、最重要的应用是互联网本身。互联网就是由大量复杂的队列构成的，这些队列具有各种不同的属性，并且使用各种复杂的方式相互连接，在这些队列中有大量的消息在移动和传递。理解和控制这样一个复杂的系统涉及队列抽象的实现、排队论的应用，以及涉及两者的模拟研究。接下来我们讨论一个经典的例子来说明这个过程。

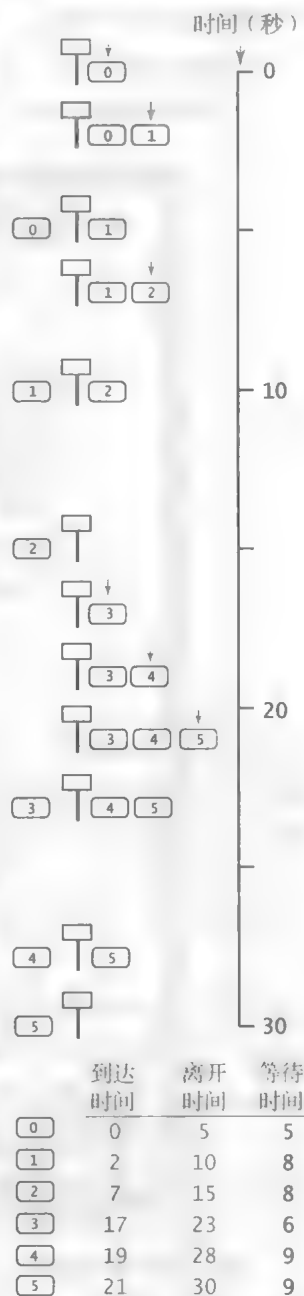
M/M/1 队列。 M/M/1 队列是最重要的队列模型之一，它被证明可以精确模拟许多现实世界的情况，比如进入收费站的一列汽车或进入急诊室的患者。M 表示马尔可夫的（Markovian）或无记忆的，表示到达和服务都是泊松过程：到达间隔时间和服务时间都服从指数分布（具体见练习 2.2.8）；1 表示有一台服务器。M/M/1 队列的参数由其到达率 λ （例如，每分钟到达收费站的车辆数）及其服务率 μ （例如，每分钟可以通过收费站的汽车数量）组成，其具有以下三个属性：

- 只有一个服务器——一个 FIFO 队列。
- 队列的到达时间间隔服从指数分布，速率为 λ / 分钟。
- 非空队列的服务时间服从指数分布，速率为 μ / 分钟。

到达的平均时间间隔是 $1/\lambda$ 分钟，平均服务时间（当队列是非空时）是 $1/\mu$ 分钟。因此，如果无法满足 $\mu > \lambda$ 则队列将无限制地增长。不然，顾客在一个有趣的动态过程中进入并离开队列。

分析。 在实际应用中，人们对参数 λ 和 μ 对队列不同属性的影响很感兴趣。如果你是一名顾客，你可能想知道在系统中预期花费的时间。如果你正在设计这个队列系统，你可能想知道有多少顾客在系统中，或者更复杂一些，比如队列大小超过给定最大容量的可能性为多少。对于简单的模型，由概率论推导出的公式可以使用 λ 和 μ 的函数来定量表示这些信息。对于 M/M/1 队列，已知：

- 系统中的平均客户数 $L = \lambda / (\mu - \lambda)$ 。
- 一名顾客在系统中花费的平均时间 $W = 1 / (\mu - \lambda)$ 。



597

一个 M/M/1 队列

例如，如果汽车以 $\lambda=10$ 辆 / 分钟的速度到达，并且服务速率是 $\mu=15$ 辆 / 分钟，则系统中汽车的平均数量将是 2，并且顾客在系统中花费的平均时间将是 1/5 分钟或 12 秒。这些公式证明，当 λ 趋近于 μ 时，等待时间（和队列长度）将会无限制地增长。它们也服从一个被称为利特尔定律的基本规则：系统中顾客的平均数量等于顾客在系统中花费的平均时间的 λ 倍 ($L=\lambda W$)。

模拟。MMIQueue（程序 4.3.7）是一个 Queue 的客户程序，可以用于来验证这些数学结果。这是一个基于事件的模拟（event-based simulation）方法的简单例子：我们生成在特定时间发生的事件，并根据事件相应地调整我们的数据结构，模拟事件发生时间点的状况。在 M/M/1 队列中，存在两种类型的事件：客户到达事件和客户服务事件。与之相对应，我们维护以下两个变量：

- nextService 是下一个服务的时间。
- nextArrival 是下一次到达的时间。

598

程序4.3.7 M/M/1队列模拟

```
public class MMIQueue
{
    public static void main(String[] args)
    {
        double lambda = Double.parseDouble(args[0]);
        double mu = Double.parseDouble(args[1]);
        Histogram hist = new Histogram(60 + 1);
        Queue<Double> queue = new Queue<Double>();
        double nextArrival = StdRandom.exp(lambda);
        double nextService = nextArrival + StdRandom.exp(mu);
        StdDraw.enableDoubleBuffering();

        while (true)
        { // 模拟服务之前到来的顾客
            while (nextArrival < nextService)
            {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }

            // 模拟服务过程
            double wait = nextService - queue.dequeue();
            hist.addDataPoint(Math.min(60, (int) Math.round(wait)));
            StdDraw.clear();
            hist.draw();
            StdDraw.show();
            StdDraw.wait(20);
            if (queue.isEmpty())
                nextService = nextArrival + StdRandom.exp(mu);
            else
                nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```

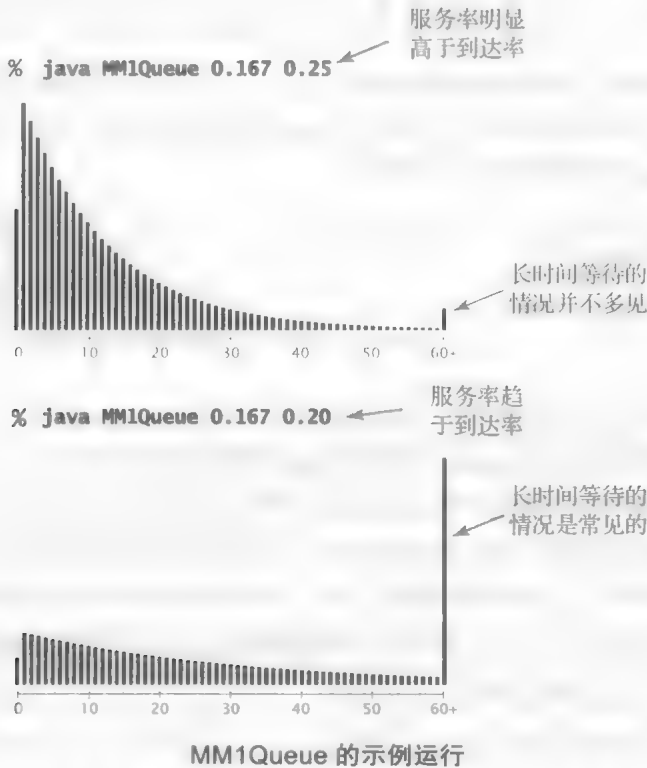
lambda	到达率
mu	服务率
hist	直方图
queue	M/M/1队列
wait	队列时间

M/M/1队列的这个模拟通过两个变量nextArrival和nextService以及一个double值队列来计算等待时间。队列中每个项的值是进入队列的时间（模拟时间）等待时间使用Histogram（程序3.2.3）进行绘制。

599

为了模拟到达事件，我们将 nextArrival（到达时间）加入队列；为了模拟一个服务，将队列中的客户移出，并根据它的到达时间计算客户的等待时间 wait（即服务完成时间减去

客户进入队列的时间)，并将等待时间加入直方图中（请参阅程序 3.2.3）。经过大量的试验，产生的形状是 M/M/1 排队系统的特征。从实际的角度来看，通过运行 MM1Queue 可以发现该过程的最重要特征之一是，对于不同的参数值 λ 和 μ ，当服务率趋近于到达率时，一名顾客在系统中花费的平均时间（以及系统中平均顾客数量）显著增加。当服务率很高时，直方图会出现一个细长的尾部，表示随着等待时间增加，给定客户需要等待时间的相应概率很快减少到可忽略。但当服务率趋于到达率时，直方图延伸到大部分值都在尾部，所以长等待时间顾客的概率占主导地位。



与我们研究过的许多其他应用程序一样，使用模拟来验证一个很好理解的数学模型是研究更复杂情况的起点。在实际队列应用程序中，我们可能有多个队列、多个服务器、多级服务器、队列长度等诸多限制。此外，到达时间间隔和服务时间的分布可能无法用数学方法描述。在这种情况下，除了使用模拟外我们别无选择。系统设计者通常建立一个队列系统的计算模型（如 MM1Queue），并使用这个计算模型来调整设计参数（如服务率），以正确响应外部环境（如到达时间）。

600

可迭代的集合 正如本节前面提到过的，数组和链表上用来处理每个元素的一个基本操作是 for 循环语句。这是一种常见的程序设计范式，并不局限于低层次的数据结构，如数组和链表。对于任何集合来说，处理其所有项的能力（可以是以某种特定的顺序）是一种非常重要的功能。客户程序的要求只是以某种方式处理每个项目，或者迭代集合中的项目。这个范式非常重要，以至于在 Java 和许多其他现代程序设计语言中都被当作最重要的使用模式来提供支持（这意味着语言本身具有支持它的特定机制，而不仅仅是通过库来支持）。有了它，我们可以编写清晰而紧凑的代码，而无须依赖集合实现的细节。

为了引入这个概念，我们从一个客户程序代码的片段开始研究，以下代码用于打印一个

字符串集合中的所有项（每行一个）：

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
    StdOut.println(s);
...
```

这个结构被称为 `foreach` 语句：你可以这样理解它：对于集合 `collection` 中的每个字符串 `s`，打印它的内容。这个客户程序代码不需要知道任何有关集合的表示或实现的细节；只是处理集合中的每一项。相同的 `foreach` 循环可以用于处理字符串队列或任何其他可迭代的字符串集合。

我们很难想象代码可以如此清晰而紧凑。但是，以这种方式实现支持迭代的集合需要一些额外的工作，我们现在详细讨论。首先，`foreach` 结构其实是一个 `while` 结构的简写。例如，前面给出的 `foreach` 语句等同于下述结构：

```
Iterator<String> iterator = collection.iterator();
while (iterator.hasNext())
{
    String s = iterator.next();
    StdOut.println(s);
}
```

601

这段代码表明了在任何可迭代集合中我们需要实现的三个必要的部分：

- 集合必须实现一个返回 `Iterator` 对象的 `iterator()` 方法。
- `Iterator` 类必须包含两个方法：`hasNext()`（返回布尔值）和 `next()`（从集合中返回一个项目）。

在 Java 中，我们使用接口继承机制来表达一个类实现一组特定方法的设计思路（参见 3.3 节）。对于可迭代集合，这些必要的接口是在 Java 语言的规范中预定义的。

为了使一个类可迭代，第一步是在其声明中添加这一句：`implements Iterable<Item>`：

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

按照以上接口（接口的定义在 `Java.lang.Iterable` 中），实现一个返回 `Iterator<Item>` 的方法并将代码添加到类中。`Iterator` 是个泛型；因此客户程序可以通过它来实现对任何指定类型对象的迭代访问（当然也仅能访问指定类型的对象）。

那么，什么是迭代器？迭代器是实现了 `hasNext()` 和 `next()` 方法等接口的类的对象，接口的定义如下（在 `Java.util.Iterator` 中定义）：

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

尽管接口需要 `remove()` 方法，但是我们在本书中总是使用一个空方法来实现 `remove()`，因为最好避免将迭代与修改数据的操作交织在一起。

在下面的两个例子中我们将展示，对于集合的数组和链表表示，实现一个迭代器类通常是很容易的。

602

使一个使用数组的类变得可迭代。对于第一个例子，我们将考虑如何使 `ArrayStackOfStrings` (程序 4.3.1) 变得可迭代。首先，将类声明更改为

```
public class ArrayStackOfStrings implements Iterable<String>
```

换句话说，我们提供一个 `iterator()` 方法，以便客户程序可以使用 `foreach` 语句遍历栈中的字符串。`Iterator()` 方法本身很简单：

```
public Iterator<String> iterator()
{ return new ReverseArrayIterator(); }
```

上述代码只是从一个实现了 `Iterator` 接口的私有嵌套类（提供了 `hasNext()`、`next()` 和 `remove()` 方法）返回一个对象：

```
private class ReverseArrayIterator implements Iterator<String>
{
    private int i = n-1;
    public boolean hasNext()
    { return i >= 0; }
    public String next()
    { return items[i--]; }
    public void remove()
    { }
}
```

请注意，嵌套类 `ReverseArrayIterator` 可以访问类内部的实例变量（这种能力是我们为迭代器使用嵌套类的主要原因），在本例中为 `items[]` 和 `n`。一个关键的细节是：在 `ArrayStackOfStrings` 代码的开始处必须添加

```
import java.util.Iterator;
```

现在，由于客户程序可以将 `foreach` 语句与 `ArrayStackOfStrings` 对象一起使用，因此可以在不知道底层数组表示的情况下遍历项。这种安排对于集合的基本数据类型的实现至关重要。例如，它使得我们可以切换到完全不同的数据实现方式，而无须更改任何客户程序代码。更重要的是，从客户程序的角度来看，它使得客户程序可以直接使用迭代，而无须了解实现的任何细节。

603

使一个使用链表的类变得可迭代。使用基本相同的步骤（当然是不同的代码）可以有效地使 `Queue` (程序 4.3.6) 可迭代，`Queue` 中原来使用的泛型不受影响。首先，将类声明更改为

```
public class Queue<Item> implements Iterable<Item>
```

换句话说，我们提供了一个 `iterator()` 方法，以便客户程序可以使用 `foreach` 语句遍历队列中的项目，而无论它们是什么类型。`iterator()` 方法本身很简单：

```
public Iterator<Item> iterator()
{ return new ListIterator(); }
```

同以前一样，我们有一个实现 `Iterator` 接口的私有嵌套类：

```
private class ListIterator implements Iterator<Item>
{
    Node current = first;
    public boolean hasNext()
    { return current != null; }
```

```

    public Item next()
    {
        Item item = current.item;
        current = current.next;
        return item;
    }
    public void remove()
    { }
}

```

同样，客户程序可以构建由任何类型的数据项构成的队列，然后遍历这些项，而无须知道底层链表的表示形式：

```

Queue<String> queue = new Queue<String>();
...
for (String s : queue)
    StdOut.println(s);

```

604 此客户程序代码更加清晰，因此比基于低级表示的代码更容易编写和维护。

我们的栈迭代器以 LIFO 顺序遍历这些项，队列迭代器以 FIFO 顺序对项进行遍历，即使并没有这些要求：我们可以以任何顺序返回数据项。但在开发迭代器时，一个明智的做法是遵循这样一个简单的规则：如果数据类型规范中希望使用某种迭代顺序，那么请使用它。

可迭代的实现可能对你来说有点复杂，但这是值得努力探究的。我们不会经常自己去实现它们，但是这样做，我们会享受到清晰和正确的客户程序代码以及代码复用带来的好处。此外，正如任何一种学过的程序设计结构一样，一旦我们能够熟练运用这些结构，你会发现自己经常使用它们并从中获益。

使一个类变得可迭代肯定会改变其 API，但为了避免过于复杂的 API 表，我们只使用形容词 *iterable* 来表示我们已将相应的代码包含到类中（如本节所述），并且客户代码可以直接使用 `foreach` 语句。从这里开始，我们将在客户程序中直接使用此处描述的可迭代（并支持泛型）的 *Stack*、*Queue* 和 *RandomQueue* 数据类型。

资源分配 接下来，我们考虑一个应用程序，演示我们已经讨论过的数据结构和 Java 语言功能。一个资源共享系统涉及大量松耦合、协作的服务器，这些服务器都同意维护自己的共享资源项的队列，一个中央管理机构将项目分发给服务器（并通知用户分配的具体位置）。例如，项目可能是被大量用户共享的歌曲、照片或视频。为强调共享的观念，我们假设系统中包括数百万个项目和数千个用户。

在这样的一个系统中，我们来考虑中央管理程序的设计，这个程序需要负责分配项目，但是忽略从系统中动态删除项、添加和删除服务器等可能的细节变动。

如果我们使用一种轮询调度策略，遍历服务器以逐个分配任务，就可以得到一个均衡的分配，但是这需要一个分配者完全控制所有的服务器，在系统中很难出现这种情况。例如，可能存在大量独立的分配者，所以没有任何一个分配者可以拥有所有服务器的最新信息。因此，这些系统常常使用随机策略，其资源分配基于随机选择。一种更好的策略是选择若干随机服务器当作一次采样，然后把一个新的项目分配给这次采样中项目数最少的服务器。对于小型分布式系统来说，这些策略之间的差异无关紧要，但对于拥有数千个服务器和数百万个项目的系统，其差异可能十分巨大，因为每个服务器都有固定数量的资源来专用于这个过程。事实上，互联网中确实存在类似的系统，其中一些队列会在特殊用途的硬件上实现，所以队列长度直接转换为额外的设备成本。但是一次采样应该抽取多大的样本比较合适呢？

```

import java.util.Iterator; ← 迭代器不在编程语言中

public class Queue<Item>
{
    implements Iterable<Item> ← 承诺实现iterator()

    private Node first;
    private Node last;
    private class Node ← 先进先出队列代码
    {
        Item item;
        Node next;
    }
    public void enqueue(Item item)
    ...
    public Item dequeue()
    ...

    public Iterator<Item> iterator() ← 实现Iterable接口
    { return new ListIterator(); } ← 使类可迭代所需要的额外代码
}

private class ListIterator
{
    implements Iterator<Item> ← 承诺实现hasNext()、next()和remove()

    Node current = first;

    public boolean hasNext()
    { return current != null; }

    public Item next() ← 实现Iterator接口
    {
        Item item = current.item;
        current = current.next;
        return item;
    }

    public void remove()
    { }
}

public static void main(String[] args)
{
    Queue<Integer> queue = new Queue<Integer>();
    while (!StdIn.isEmpty())
        queue.enqueue(StdIn.readInt());
    for (int s : queue) ← foreach语句
        StdOut.println(s);
}

```

解析一个迭代类

LoadBalance (程序 4.3.8) 是一个采样策略的模拟，可以用于研究这个问题。程序充分利用了数据结构（队列和随机队列）和高级结构（泛型和迭代器）这些我们已经仔细分析过的数据结构，因此实验程序并不难理解。模拟程序维护了一个随机队列，并在内循环中完成计算，使用 RandomQueue 的 sample() 方法（练习 4.3.36）随机抽样队列，并将新的服务请求分配到最短的队列中。令人惊讶的是，实验结果表明，大小为 2 的采样会得到近乎完美的平衡，所以扩大采样没有意义。

我们详细讨论了有关栈和队列 API 的基本实现，特别是分析了不同实现所需要的空间和时间。我们这么做不仅是因为这些数据类型十分重要和有用，而且还因为你在自己定义数据类型和实现过程中极有可能遇到相同的问题。

在开发一个用于维护数据集合的客户程序时，我们应该使用下推栈、FIFO 队列还是随机队列？这个问题的答案取决于对客户需求的深层次分析，以确定 LIFO、FIFO 或随机队列哪一种规则更合适。

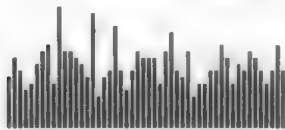
程序4.3.8 负载均衡模拟

```
public class LoadBalance
{
    public static void main(String[] args)
    { // 将n项分配到m个服务器上, 优先分配给
      // 一次采样中队列最短的服务器
      int m = Integer.parseInt(args[0]);
      int n = Integer.parseInt(args[1]);
      int size = Integer.parseInt(args[2]);
      // 创建服务器队列
      RandomQueue<Queue<Integer>> servers;
      servers = new RandomQueue<Queue<Integer>>();
      for (int i = 0; i < m; i++)
          servers.enqueue(new Queue<Integer>());
      for (int j = 0; j < n; j++)
      { // 将一个任务项分配到一个服务器上
        Queue<Integer> min = servers.sample();
        for (int k = 1; k < size; k++)
        { // 从随机选取的服务器中取出一个, 如果它是最小的, 那么就更新min
          Queue<Integer> queue = servers.sample();
          if (queue.size() < min.size()) min = queue;
        } // min中保存的服务器队列最短
        min.enqueue(j);
      }
      int i = 0;
      double[] lengths = new double[m];
      for (Queue<Integer> queue : servers)
          lengths[i++] = queue.size();
      StdDraw.setYscale(0, 2.0*n/m);
      StdStats.plotBars(lengths);
    }
}
```

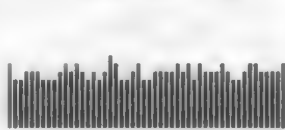
m	服务器数目
n	项目数
size	采样大小
servers	队列
min	采样中最短队列
queue	当前服务器

这个程序使用了Queue和RandomQueue的泛型, 模拟了将n项目分配给由m个服务器组成的服务器集合的过程。请求被放入随机选择的一批服务器中队列最短的服务器上。

% java LoadBalance 50 500 1



% java LoadBalance 50 500 2



我们应该使用数组、链表还是可变数组组织数据呢? 这个问题的答案依赖于对性能需求的分析, 这通常是低层次的分析。数组的优点是可以在常量时间内访问任何项目, 但缺点是需要事先知道其最大长度。链表的优点是可以容纳的项目数量没有限制, 缺点是不能在常量时间内访问所需的元素。可变数组结合了数组和链表的优点 (可以在常量时间内访问任何元素, 且无须事先知道最大长度), 但可变数组有一个 (轻微的) 缺点, 就是运行时间只能在摊销的基础上才能算作常量 (对于某一次单独的数据项访问, 无法确定所需的时间——译者注)。每种数据结构都只能适用于特定的场景; 在很多编程环境中, 你都会看到对这三种数据结构的应用。例如, Java 类 Java.util.ArrayList 中使用了可变数组, Java 类 Java.util.LinkedList 中使用了链表。

本节中我们讨论过的强大的高级构造和新的语言特性 (泛型和迭代器) 并不是一开始就有的。它们是复杂的编程语言特性, 直到 21 世纪初才被广泛使用, 但仍然只是主要由专业程序员使用。尽管如此, 这些新特性的使用正在飞速发展, 一方面, 传统的 Java 和

C++ 都提供了完善的支持，而像 Python 和 Ruby 这样的新语言也支持这些高级构造和语言特性；另一方面，掌握了这些新的特性确实可以提高编程的效率和质量。现在，我们已经知道了，学习使用一个新的语言特性并没有比学习骑自行车或学习编写 HelloWorld 程序困难多少：一开始看起来非常神秘，直到你完成了第一次尝试后发现没什么难的，然后很快它会成为你的第二本能。学会如何使用泛型和迭代器是一件非常值得我们花时间完成的事情。

607
608

问答环节

问：我应该在什么时候通过 new 创建 Node 对象？

答：与任何其他类一样，当我们想要创建一个新的 Node 对象（链表中的新节点）时应该使用 new。我们不应该使用 new 来创建一个对现有 Node 对象的引用。例如，代码

```
Node oldFirst = new Node();
oldFirst = first;
```

创建一个新的节点对象，然后指向它的唯一引用又立即丢失了。上述代码不会产生错误，但毫无理由地创建孤立项是不规范的做法。

问：为什么要将 Node 声明为嵌套类？为什么是私有的？

答：把嵌套类 Node 声明为私有的，那么类中的方法可以引用 Node 对象，但是在其他类中无法访问。说明：非静态的嵌套类被称为内部类，所以技术上说 Node 类是内部类，尽管非泛型的 Node 类可以被声明是静态的。

问：当键入 Javac LinkedStackOfStrings.java 来运行程序 4.3.2 和类似的程序时，除了 LinkedStackOfStrings.class 之外，还能找到一个文件 LinkedStackOfStrings \$ Node.class。这个文件的作用是什么？

答：这是嵌套类 Node 生成的文件。Java 的命名约定是使用 \$ 符号将外部类的名称与嵌套类分开。

问：是否允许客户程序将 null 项目插入栈或队列？

答：在 Java 中实现集合数据结构时这个问题经常出现。在我们的实现（以及 Java 库的栈和队列）中，允许插入 null。

问：是否有栈和队列的 Java 库？

答：是也不是。Java 有一个名为 Java.util.Stack 的内置库，但是当你需要一个栈时，你应该尽量不要使用它。这个库有几个额外的操作，与常用的栈的功能并不完全相符。例如，它可以获取第 i 个项目，也允许添加一个项目到栈的底部（而不是顶部），所以它可以实现队列的功能！虽然有这样的额外操作似乎是一个惊喜，但实际上也是一个诅咒。我们使用数据类型不是因为它们提供了所有可用的操作，而是因为它们仅仅允许了指定所需的操作。这样做的主要好处是系统可以阻止我们执行实际上不必要的操作。Java.util.Stack 的 API 是一个过于宽泛的接口，我们通常应该尽量避免。

609

问：我希望对泛型栈使用数组表示形式，但像下面这样的代码不能通过编译。有什么问题吗？

```
private Item[] item = new Item[capacity];
```

答：这是一个好的尝试。遗憾的是，Java 不允许创建泛型数组。专家们仍在积极地争

论这一决定。像通常一样，对编程语言特性的抱怨太多会让我们逐渐变成一个编程语言设计师。解决这个问题有一个办法，使用强制转换。你可以这样写：

```
private Item[] item = (Item[]) new Object[capacity];
```

问：为什么导入的类是 `Java.util.Iterator` 而不是 `Java.lang.Iterable`？

答：这是历史原因，接口 `Iterator` 是 `Java.util` 包的一部分，默认情况下不导入。接口 `Iterable` 是相对较新的，且作为 `Java.lang` 包的一部分，会被默认导入。

问：请问可以在数组中使用 `foreach` 语句吗？

答：可以（即使在技术上，数组没有实现 `Iterable` 接口）。以下代码用于将命令行参数打印到标准输出：

```
public static void main(String[] args)
{
    for (String s : args)
        StdOut.println(s);
}
```

610

问：请问在使用泛型时，如果在声明或构造函数调用中省略 `type` 参数，会发生什么情况？

```
Stack<String> stack = new Stack();           // 不安全
Stack          stack = new Stack<String>();  // 不安全
Stack<String> stack = new Stack<String>();    // 正确
```

答：第一个语句会产生编译时警告。第二个语句不直接产生警告，但是如果使用 `String` 参数调用 `stack.push()`，并且将 `stack.pop()` 的结果分配给 `String` 类型的变量，则会产生编译时警告。作为第三个语句的替代，我们可以使用钻石运算符（diamond operator），它允许 `Java` 从上下文推断构造函数调用的类型参数：

```
Stack<String> stack = new Stack<>(); // 钻石运算符
```

问：为什么不能有个 `Collection` 数据类型，可以添加项、删除最近插入的项、移除最早插入的项、删除一个随机项、迭代访问这些项、返回集合中的项数，以及任何我们可能需要的其他操作？然后，我们可以让它们都实现在一个类中，可以被许多客户程序使用。

答：这是一个宽泛接口（wide interface）的例子，正如我们在 3.3 节中指出的那样，我们应该避免这样的编程。避免宽泛接口的一个原因是很难构造对所有操作都有效的实现。一个更重要的原因是，窄接口可以强调程序的某些规则，从而使得客户程序代码更容易理解。如果一个客户程序使用 `Stack<String>`，而另一个使用 `Queue<Customer>`，我们很容易理解第一个程序强调 LIFO 规则，而第二个程序强调 FIFO 规则。另一个方法是使用继承尝试封装对所有集合都通用的操作。然而，我们建议此类实现交给专家们去完成，程序员则可以学习构建通用数据类型实现，如 `Stack` 和 `Queue`。

611

练习

4.3.1 将一个方法 `isFull()` 添加到 `ArrayStackOfStrings`（程序 4.3.1），如果栈大小等于数组容量，则返回 `true`。修改 `push()` 使其在栈已满时引发异常。

4.3.2 给出“Java `ArrayStackOfStrings` 5”命令针对以下输入的输出结果：

```
it was - the best - of times - - - it was - the - -
```

4.3.3 假设一个客户程序在一个栈上执行了一系列 push 入栈和 pop 出栈的混合操作。入栈操作把整数 0 至 9 按顺序压入栈；出栈操作输出返回的结果值。下列序列中，哪一个不可能出现？

- | | |
|------------------------|------------------------|
| a. 4 3 2 1 0 9 8 7 6 5 | b. 4 6 8 7 5 3 2 9 0 1 |
| c. 2 5 6 7 4 8 9 3 1 0 | d. 4 3 2 1 0 5 6 7 8 9 |
| e. 1 2 3 4 5 6 9 8 7 0 | f. 0 4 6 5 3 8 1 7 2 9 |
| g. 1 4 7 9 8 6 5 3 0 2 | h. 2 1 4 3 6 5 8 7 9 0 |

4.3.4 编写一个栈的客户程序 Reverse，从标准输入读取字符串，并按相反顺序写入到标准输出。使用栈或队列。

4.3.5 编写一个静态方法，从标准输入中逐个读取若干个浮点数，一次一个，并按照它们在标准输入上出现的顺序返回一个包含它们的数组。提示：使用堆栈或队列。

4.3.6 编写一个栈的客户程序 Parentheses，从标准输入中读取一串小括号、方括号和大括号的括号字符串，并使用栈来确定它们是否正确配对。例如，对于 “[()] {} {[()]}()” 程序结果为 true；对于 “[()]” 程序结果为 false。提示：使用栈。

612

4.3.7 请问当 n 为 50 时，下面的代码片段会打印什么？给定正整数 n，请描述如下代码片段所实现的功能。

```
Stack<Integer> stack = new Stack<Integer>();
while (n > 0)
{
    stack.push(n % 2);
    n /= 2;
}
while (!stack.isEmpty())
    StdOut.print(stack.pop());
StdOut.println();
```

答案：该代码打印 n 的二进制表示（n 为 50 时，结果为 110010）。

4.3.8 下面的代码片段对队列 queue 的操作结果是什么？

```
Stack<String> stack = new Stack<String>();
while (!queue.isEmpty())
    stack.push(queue.dequeue());
while (!stack.isEmpty())
    queue.enqueue(stack.pop());
```

4.3.9 将方法 peek() 添加到 Stack（程序 4.3.4）中，用于返回栈中最近插入的项目（但不删除这个项目）。

4.3.10 使用如下输入，程序 ResizingArrayStackOf 每次操作后数组的内容和长度分别是什么？

it was - the best - of times - - - it was - the - -

4.3.11 向 Stack（程序 4.3.4）和 Queue（程序 4.3.6）中添加方法 size()，返回集合中数据项的数量。提示：维护一个实例变量 n，初始化为 0；n 在 push() 和 enqueue() 时递增，在 pop() 和 dequeue() 时递减，在调用 size() 时返回其值，确保你采用的方法可以在常量时间内运行。

613

4.3.12 按 4.1 节中图表的样式绘制一个内存使用图，用于描述本节介绍链表时所使用的三个节点示例。

4.3.13 编写一个程序，从标准输入中接收一个没有左括号的表达式，输出相应的中缀表达式，注意适当插入配对的括号。例如，对于给定的输入：

1 + 2) * 3 - 4) * 5 - 6)))

程序应该输出：

((1 + 2) * ((3 - 4) * (5 - 6)))

614

- 4.3.14 编写一个过滤器程序 `InfixToPostfix`，将算术表达式从中缀形式转换为后缀形式。
- 4.3.15 编写一个程序 `EvaluatePostfix`，程序从标准输入读取一个后缀表达式，对该表达式进行计算并在标准输出上显示结果（将前一个题目的程序输出连接到本程序，其行为等价于程序 4.3.5 中的 `Evaluate`）。
- 4.3.16 假设一个客户程序在一个 FIFO 队列上执行一系列 `enqueue` 和 `dequeue` 操作。`enqueue` 操作将整数 0 到 9 依次插入队列中；`dequeue` 操作输出返回的结果值。请问下列哪一个序列是不可能出现的？
- a. 0 1 2 3 4 5 6 7 8 9

b. 4 6 8 7 5 3 2 9 0 1

c. 2 5 6 7 4 8 9 3 1 0

d. 4 3 2 1 0 5 6 7 8 9
- 4.3.17 编写一个可迭代 `Stack` 的客户程序，使用静态方法 `copy()`，将一个字符串栈作为参数并返回这个栈的一个副本。练习 4.3.48 中还有这个问题的另一种解法。
- 4.3.18 编写一个 `Queue` 的客户程序，从命令行接收一个参数 `k`，并从标准输入获取一个字符串，在该字符串中查找并输出倒数第 `k` 个字符。
- 4.3.19 开发一个数据类型 `ResizingArrayQueueOfStrings`，首先基于一个固定长度的数组实现一个队列，使得所有的操作时间复杂度都是一样的。然后，扩展这个实现使用可变数组来摆脱长度限制。提示：难点是当项目被添加到队列中或从队列中移除时，项目会在数组中“折回”。请使用算术取模来维护记录队列前端和后端项的数组索引下标。

StdIn	StdOut	n	lo	hi	items[]							
					0	1	2	3	4	5	6	7
		0	0	0	null							
to		1	0	1	to	null						
be		2	0	2	to	be						
or		3	0	3	to	be	or	null				
not		4	0	4	to	be	or	not				
to		5	0	5	to	be	or	not	to	null	null	null
-	to	4	1	4	null	be	or	not	to	null	null	null
be		5	1	6	null	be	or	not	to	be	null	null
-	be	4	2	6	null	null	or	not	to	be	null	null
-	or	3	3	6	null	null	null	not	to	not	null	null
that		4	3	7	null	null	null	not	to	not	that	null

615

- 4.3.20（这个问题倾向于数学题）证明 `ResizingArrayStackOfStrings` 中的数组永远不会少于四分之一满。然后证明，对于任何 `ResizingArrayStackOfStrings` 客户程序来说，所有栈操作的总成本除以操作次数都是在一个常量限度内。
- 4.3.21 修改 `MM1Queue`（程序 4.3.7）为程序 `MD1Queue`，在新程序中模拟一个队列，其服务时间采用固定（确定）速率 μ 。针对这个模型，验证利特尔定律。
- 4.3.22 开发一个类 `StackOfInts`，使用链表表示（不使用泛型）来实现整数栈。并编写一个客户程序，将实现的性能与 `Stack<Integer>` 进行比较，以确定系统中自动装箱和拆箱的性能损失。
- 4.3.23 假设 `x` 是一个链表节点。请问下列代码片段的作用是什么？
- ```
x.next = x.next.next;
```
- 答案：从列表中删除紧跟在 `x` 之后的一个节点。
- 4.3.24 编写一个方法 `find()`，将链表中的第一个节点和一个字符串 `key` 作为参数，如果链表中某个节点的数据项值是 `key`，则返回 `true`，否则返回 `false`。

4.3.25 编写一个方法 delete(), 该方法将链表中的第一个节点和一个整数 k 作为参数, 删除链表中的第 k 个元素 (如果存在的话)。

4.3.26 假设 x 是一个链表节点。以下代码片段的作用是什么?

```
t.next = x.next;
x.next = t;
```

答案: 在节点 x 之后立即插入节点 t。

4.3.27 为什么下面的代码片段与上一个问题中的代码片段没有相同的效果?

```
x.next = t;
t.next = x.next;
```

答案: 当执行到更新 t.next 的时候, x.next 已经不再是 x 后面的原始节点, 而是 t 本身!

4.3.28 编写一个方法 removeAfter(), 参数为一个链表节点, 然后移除给定的节点的下一个节点 (如果参数为 null 或参数的下一个字段为 null, 则不执行任何操作)。

616

4.3.29 编写一个方法 copy(), 参数为一个链表节点, 创建一个新的链表, 包含与给定链表具有相同项目的序列, 注意保持原始链表不变。

4.3.30 编写一个方法 remove(), 参数为一个链表的节点和一个字符串 key, 删除链表中所有项目为 key 的节点。

4.3.31 编写一个方法 max(), 将链表中的第一个节点作为参数, 返回列表中最大项的值。假设所有项都是正整数, 如果链表为空则返回 0。

4.3.32 针对上一个问题开发一个递归的解决方案。

4.3.33 编写一个方法, 将链表中的第一个节点作为参数, 反转列表, 并返回结果链表中的第一个节点。

4.3.34 编写一个递归方法, 反向输出链表中的项目。请不要修改链表中的任何链接。简单方法: 使用二次型运行时间, 加上常量型额外空间。另外一种简单的方法: 使用线性运行时间, 加上线性额外的空间。复杂方法: 开发一个需要线性时间并使用对数额外空间的分而治之的算法。

4.3.35 编写一个递归方法, 通过修改链接的方式随机混洗链表的节点。简单方法: 使用二次型运行时间, 常量型额外空间。复杂方法: 开发一个分而治之算法, 需要线性对数型运行时间和对数型额外空间。有关“合并”步骤请参见练习 1.4.40。

617

创新练习

4.3.36 Deque。Deque (发音为 “deck”) 是一个双端队列, 是一个由栈和队列组成的集合。编写一个 Deque 类, 使用链表实现下列 API:

|                          |   |            |
|--------------------------|---|------------|
| public class Deque<Item> |   |            |
| Deque()                  | : | 新建一个空的双端队列 |
| boolean isEmpty()        | : | 双端队列为空?    |
| void enqueue(Item item)  | : | 向队尾添加项     |
| void push(Item item)     | : | 向队首添加项     |
| Item pop()               | : | 在队首处移除并返回项 |
| Item dequeue()           | : | 在队尾处删除并返回项 |

双端队列的泛型 API

4.3.37 随机队列。随机队列是支持以下 API 的集合:

```
public class RandomQueue<Item>
```

|                                      |                        |
|--------------------------------------|------------------------|
| <code>RandomQueue()</code>           | 新建一个空的随机队列             |
| <code>boolean isEmpty()</code>       | 随机队列为空?                |
| <code>void enqueue(Item item)</code> | 向随机队列中添加项              |
| <code>Item dequeue()</code>          | 取出并返回一个随机项 (不重置采样)     |
| <code>Item sample()</code>           | 返回一个随机项目, 但不要删除 (重置采样) |

### 随机队列的泛型 API

编写一个类 `RandomQueue` 实现这个 API。提示: 使用可变数组。要删除一个项目, 将随机位置 (索引下标为 0 到  $n-1$ ) 的项与最后位置 (索引  $n-1$ ) 的项交换, 然后删除并返回随机队列的最后一个项, 如 `ResizingArrayStack` 中所示。编写一个客户程序, 使用 `RandomQueue<Card>` 输出一副随机顺序的扑克牌。

[618]

- 4.3.38 随机迭代器。为上一个练习中的 `RandomQueue<Item>` 编写一个迭代器, 以随机顺序返回项目。不同的迭代器应该以不同的随机顺序返回项目。注意: 这个练习做起来比看起来困难。
- 4.3.39 约瑟夫 (Josephus) 问题。约瑟夫问题是一个古老的问题, 在这个问题中有  $n$  个人陷入困境, 一致同意按以下策略减少人口。他们排列成一个圈 (从 0 到  $n-1$  的位置), 然后由第一个人开始按照圆圈报数, 每报数到第  $m$  个人就杀掉这个人, 然后再由下一个人重新报数, 直到只剩下最后一个人。据说约瑟夫想知道坐哪里可以避免被杀死。便编写一个 `Queue` 的客户程序 `Josephus`, 从命令行接收两个整数参数  $m$  和  $n$ , 并输出人们被杀死的顺序 (从而确定约瑟夫坐在圆圈中的位置)。

```
% java Josephus 2 7
1 3 5 0 4 2 6
```

- 4.3.40 广义队列。实现一个支持以下 API 的类, 通过支持删除第  $i$  个最近插入的项来泛化队列和栈:

```
public class GeneralizedQueue<Item>
```

|                                  |                 |
|----------------------------------|-----------------|
| <code>GeneralizedQueue()</code>  | 创建一个空的广义队列      |
| <code>boolean isEmpty()</code>   | 广义队列是空的吗        |
| <code>void add(Item item)</code> | 将item插入广义队列     |
| <code>Item remove(int i)</code>  | 删除并返回第i个最近插入的项目 |
| <code>int size()</code>          | 队列中的项目数         |

### 通用广义队列的 API

首先, 开发一个使用可变数组的实现, 然后开发一个使用链表的实现 (有关使用二叉查找树的更为有效的实现请参见练习 4.4.57)。

[619]

- 4.3.41 环形缓冲区。环形缓冲区 (或循环队列) 是一个 FIFO 集合, 用于存储一系列项, 其中项数的上限是已知的。如果将项插入已满的环形缓冲区中, 新项会替换最早插入的项。环形缓冲区可用于在异步进程间传输数据和存储日志文件。当缓冲区为空时, 用户等待直到数据存入其中; 当缓冲区为满时, 生产者暂停存放数据。为环形缓冲区开发一个 API, 并给出一个基于数组表示的实现。
- 4.3.42 合并两个有序的队列。给定两个按升序排列的字符串序列, 将所有字符串移动到第三个字符串序列中, 同时保证第三个队列的结果也按升序排列。

- 4.3.43 非递归的归并排序。给定  $n$  个字符串，创建  $n$  个队列，每个队列包含一个字符串。创建一个包含  $n$  个队列的队列。然后，应用前一个练习中的排序合并操作将队列中的前两个队列合并，并将合并后的队列插入尾端。重复此过程直到这个队列的队列只包含一个队列。
- 4.3.44 双栈队列。请使用两个栈实现一个队列。提示：如果将项压栈，然后全部弹出，则它们将以相反的顺序出现。重复这个过程使得它们回到 FIFO 顺序。
- 4.3.45 移动到前端 (Move-to-front)。从标准输入中读取一系列字符，把字符保存在一个链表中，假设字符没有重复。当读取一个新的字符时，将其插入链表的前端。当读取一个重复字符时，从链表中删除该字符并重新插入链表的开头。这样，程序就实现了众所周知的移动到前端策略。这个策略可以用于缓存、数据压缩以及许多其他应用程序，因为一个最近被访问的项目更有可能被重新访问。
- 4.3.46 拓扑排序。服务器上编号从 0 到  $n-1$  的  $n$  个作业需要进行排序。有些工作必须先完成，其他工作才能开始。编写一个程序 TopologicalSorter，它接收一个命令行参数  $n$  和标准输入的一系列有序的作业对 “ij”，然后输出一个整数序列，使得对于输入的每一个作业对 “ij”，作业  $i$  总出现在作业  $j$  之前。使用以下算法：首先，从输入中为每个作业构建：① 一个必须跟随该作业的作业队列；② 这个节点的入度（必须在其之前完成的作业数）。然后，构建一个所有入度为 0 的节点的队列，重复删除入度为 0 的所有作业，并更新所有的数据结构。这个过程有很多实际应用。例如，可以使用它来为你的专业建立课程排序（使得你开始修读某门课时，它所需要的前序课程都已修读过——译者注），以便你可以选修一系列课程，从而顺利毕业。
- 4.3.47 文本编辑缓冲区。请开发一个数据类型，用于文本编辑器的缓冲区，实现如下 API：

620

| public class Buffer |             |
|---------------------|-------------|
| Buffer()            | 创建一个空的缓冲区   |
| void insert(char c) | 在光标位置插入c    |
| char delete()       | 删除并返回光标处的字符 |
| void left(int k)    | 将光标k的位置向左移动 |
| void right(int k)   | 将光标k的位置向右移动 |
| int size()          | 缓冲区中的字符数    |

文本缓冲的 API

提示：使用两个栈。

- 4.3.48 复制堆栈的构造函数。为 Stack 的链表实现创建一个新的构造函数：

```
Stack<Item> t = new Stack<Item>(s);
```

以上代码使得  $t$  引用队列  $s$  的一个新的独立拷贝。这一句完成后，从  $s$  或  $t$  进行压入和弹出操作应该不影响另一方。

- 4.3.49 复制队列的构造函数。创建一个新的构造函数：

```
Queue<Item> r = new Queue<Item>(q);
```

使得队列  $r$  成为对队列  $q$  的新的独立拷贝的引用。

621

- 4.3.50 Quote。开发一个数据类型 Quote，用于实现以下 API：

```
public class Quote
```

|                                           |                  |
|-------------------------------------------|------------------|
| <code>Quote()</code>                      | 创建一个空的Quote      |
| <code>void add(String word)</code>        | 将word附加到Quote的末尾 |
| <code>void add(int i, String word)</code> | 在索引处插入word       |
| <code>String get(int i)</code>            | 返回索引的单词          |
| <code>int count()</code>                  | Quote中的单词数       |
| <code>String toString()</code>            | Quote中的单词        |

#### quote 的 API

为此，请定义一个嵌套类 Card，它包含引文中的一个单词和指向引文中下一个单词的链接：

```
private class Card
{
 private String word;
 private Card next;
 public Card(String word)
 {
 this.word = word;
 this.next = null;
 }
}
```

- 4.3.51 循环 quote。重复上一个练习，但使用循环链表。在循环链表中，每个节点指向其后继，列表中的最后一个节点指向第一个节点（而不是 null，标准链表中最后一个节点指向 null）。
- 4.3.52 链表反向（迭代）。编写一个非递归函数，它将链表中的第一个节点作为参数，并反转列表，返回结果中的第一个节点。
- 4.3.53 链表反向（递归）。编写一个递归函数，它将链表中的第一个节点作为参数，并反转列表，返回结果中的第一个节点。
- 4.3.54 队列模拟。将 MM1Queue 改为使用栈实现，而不是队列，会发生什么？利特尔定律是否依旧成立？针对一个随机队列，请回答相同的问题。绘制直方图，比较等待时间的标准偏差。
- 4.3.55 负载均衡模拟。修改 LoadBalance，输出平均队列长度和最大队列长度（而不是绘制直方图），并使用修改后的程序来运行在 100 000 个队列上分配 100 万个项目的情况，输出 100 次试验（每次采样大小为 1、2、3 和 4）的最大队列长度的平均值。请问你的实验是否验证了正文中使用采样大小为 2 的结论？
- 4.3.56 文件列表。文件夹是文件和文件夹的列表。编写一个程序，将文件夹的名称作为命令行参数，输出该文件夹下包含的所有文件，每个文件夹的内容在该文件夹的名称下递归列出（采用缩进方式）。提示：使用队列，参考 Java.io.File。

## 4.4 符号表

符号表（symbol table）是一种数据结构，用于将数值与关键字关联。客户程序可以通过指定一个键值对将一个项目存储到（put）符号表中，然后从符号表中检索（get）对应于特定键的值。例如，一所大学可能把一个学生的姓名、家庭地址、成绩等（值）与该学生的社会安全号码（键）相关联，以便通过指定的社会安全号码来访问每个学生的记录信息（社会安全号码为美国社会的唯一标识，类似我国的身份证号码——译者注）。同样的方法适用于科学家组织数据、企业跟踪客户交易信息、Internet 搜索引擎关联关键字和网页，或者其他许多应用场景。



在本节中，我们讨论符号表数据类型的基本 API。除 put 和 get 操作外，我们的 API 还包括测试是否存在值与一个给定键相关联（contains）、移除键（以及相关关联的值）、确定符号表中键值对的数量（size），以及对符号表中的键遍历访问的功能。我们也会讨论在各种应用中自然产生的符号表上的其他基于元素顺序的操作。

作为动力，我们讨论两个典型的客户程序——字典查找和索引，并简要地讨论它们在实际中的应用。类似的客户程序是基本的工具，它们经常以某种形式在每个计算环境中呈现，容易当成理所当然的而忽略其原理，也容易产生误用。同任何复杂的工具一样，想要高效地使用字典或者索引，必须了解它们是如何构建的。这就是我们在本节中详细研究符号表的原因。

由于其本质上的重要性，符号表自计算初期就被广泛使用和研究。我们讨论两个经典实现。第一个是散列操作，用于将键转换为数组索引下标以访问值。第二种是基于二叉搜索树（BST）的数据结构。两个实现都是非常简单的解决方案，在许多应用场景中性能良好，可以作为现代程序设计环境中工业级符号表实现的基础。我们认为散列和二叉搜索树的代码比我们讨论过的栈和队列中的链表代码稍微复杂一点，但它将会引领你进入一个具有深远影响的数据结构的新维度。

624

**API** 符号表（symbol table）是一个键 - 值对的集合。我们使用泛型 key 表示键和泛型 Value 来表示值。每一个符号表项保存一个值与一个键的关联。这些假设即可得出以下几个基本 API：

|                                                 |  |              |
|-------------------------------------------------|--|--------------|
| <code>public class *ST&lt;Key, Value&gt;</code> |  |              |
| <code>*ST()</code>                              |  | 创建一个空的符号表    |
| <code>void put(Key key, Value val)</code>       |  | 将val与key建立关联 |
| <code>Value get(Key key)</code>                 |  | 获取与key相关联的值  |
| <code>void remove(Key key)</code>               |  | 删除键key及其对应的值 |
| <code>boolean contains(Key key)</code>          |  | 是否存在对应于key的值 |
| <code>int size()</code>                         |  | 键-值对的数量      |
| <code>Iterable&lt;Key&gt; keys()</code>         |  | 返回符号表中所有的键   |

### 泛型化符号表的 API

同之前一样，“\*”是一个占位符，表明可以考虑多种实现。在本节中，我们提供了两个经典实现：HashST 和 BST（我们在文中也简要介绍了一些基本的实现）。这个 API 反映了若干设计选择，我们将逐一展开讨论。

**不可变键。**我们假设键在符号表中保持值不变。最简单和最常用的键类型包括 String 类型和内置包装类型（如 Integer 和 Double），并且通常假设键的值是不可变的。

**替换旧值策略。**如果要将一个值关联到一个已经包含关联值的键，我们采用新值替换旧值的一贯原则（就像使用赋值语句给数组元素赋值）。如果需要，contains() 方法为客户提供了避免这种操作的灵活性。

625

**不存在。**如果符号表中不存在与指定键相关的值，get() 方法返回 null。这个选择有两个含义，接下来讨论。

**空键和空值。**不允许客户将 null 用作键或值。这个规定使我们能够按照如下所示实现 contains()：

```
public boolean contains(Key key)
{ return get(key) != null; }
```

移除。我们讨论的符号表 API 中还包含从符号表中删除键（及其相关联的值）的方法，因为许多应用程序确实需要这种方法。但是，简便起见，我们将移除功能的实现推迟到练习或算法和数据结构这样更高级的课程中。

遍历键 - 值对。keys() 方法为客户提供了遍历数据结构中键 - 值对的方法。为了简便，它只返回键；如果需要，客户可以使用 get 来获取键相关联的值。客户程序代码如下所示：

```
ST<String, Double> st = new ST<String, Double>();
...
for (String key : st.keys())
 StdOut.println(key + " " + st.get(key));
```

散列键。像许多语言一样，Java 为符号表的实现直接提供了所需的语言和系统支持。具体而言，每种类型的对象都有一个 equals() 方法（我们可以使用这个方法测试两个键是否相同，正如键数据类型所定义的那样）和一个 hashCode() 方法（它支持特定类型的符号表的实现，我们将在本节稍后讨论）。对于最常用的键的标准数据类型，我们可以依赖于这些方法的系统实现。相反，对于我们创建的数据类型，我们必须仔细考虑实现，如 3.3 节所讨论的。大多数程序员只是简单地假设所有需要的功能都已被妥善实现，但实际上并不总是这样，因此在处理自定义的键类型时需要小心。

可比较的键。在许多应用程序中，键可以是字符串或其他具有自然顺序的数据类型。在 Java 中，如 3.3 节所述，我们期望这些键实现 Comparable 接口，从而是可比较的。具有可比较的键的符号表非常重要，其原因有两个。首先，我们可以利用键的顺序来开发 put 和 get 的实现，从而保证 API 中的性能符合规范要求。其次，基于可比较的键可以定义大量的新功能（并且也不难实现）。一个客户程序也许需要最小键、最大键、中间键或按某种顺序遍历所有键。算法和数据结构的书籍中会更加全面而深入的探讨该问题，在本节稍后部分，你会学习一个简单的数据结构，可以轻松地实现以下显示的 API 中详细介绍的操作。

626

| public class *ST<Key extends Comparable<Key>, Value> |  |                    |
|------------------------------------------------------|--|--------------------|
| *ST()                                                |  | 创建一个空的符号表          |
| void put(Key key, Value val)                         |  | 将val与key建立关联       |
| Value get(Key key)                                   |  | 获取与key相关联的值        |
| void remove(Key key)                                 |  | 删除键key及其对应的值       |
| boolean contains(Key key)                            |  | 是否存在对应于key的值       |
| int size()                                           |  | 键值对的数量             |
| Iterable<Key> keys()                                 |  | 按照排好序的顺序返回符号表中所有的键 |
| Key min()                                            |  | 返回最小的键             |
| Key max()                                            |  | 返回最大的键             |
| int rank(Key key)                                    |  | 小于key的键的数量         |
| Key select(int k)                                    |  | 从小到大排序的第k个键        |
| Key floor(Key key)                                   |  | 小于等于key的最大键        |
| Key ceiling(Key key)                                 |  | 大于等于key的最小键        |

有序符号表的 API

符号表是计算机科学中研究最为广泛的数据结构之一，所以本文中所讨论的符号表的内容和许多其他替代设计的影响一直以来都被细致地研究，如果你学习计算机科学的后续课程，将会了解到更多这方面的知识。在本节中，我们的方法是通过考虑两个典型的客户程序来介绍符号表最重要的属性，开发两种经典和高效的实现，并研究两种实现的性能特性，以

证明它们能够满足典型客户程序的需求（即使在符号表巨大的情况下）。 [627]

**符号表客户程序** 一旦你获得了有关符号表的一些理念和体验，你会发现符号表的用途十分广泛。为了证明这个事实，我们从两个典型的例子开始，它们均可用于大量重要和熟悉的实际应用程序。

字典查找。最基本的符号表客户程序通过连续的 put 操作来构建符号表，以支持 get 请求。我们维护一组数据以便需要时可以快速访问数据。大多数应用程序还充分利用了“符号表是一个动态字典”的思想，不仅容易查找表中信息，而且可以方便地更新表中信息。如下例子说明了这种方法的实用性。

|              | 键     | 值    |
|--------------|-------|------|
| 电话簿          | 姓名    | 电话号码 |
| 字典           | 单词    | 释义   |
| 账户信息         | 账户号   | 存款值  |
| 基因组学         | 密码子   | 氨基酸  |
| 实验数据         | 数据/时间 | 结果   |
| 编译器          | 变量名   | 内存地址 |
| 文件共享服务       | 歌曲名   | 主机名  |
| 互联网域名解析（DNS） | 网址    | IP地址 |

典型的字典应用

- **电话簿。**以人名为键，以其电话号码为值，符号表就模拟了一个电话簿。与印刷电话簿的主要区别在于我们可以添加新的姓名或更改现有的电话号码。我们也可以使用电话号码作为键，姓名作为相应的值。如果你从未这样尝试过，请试着在浏览器搜索框中输入你的电话号码（带区号）。 [627]
- **词典。**把一个单词与其释义联系起来就是我们常见的“词典”。几个世纪以来，人们在家里和办公室中都放置了印刷的词典，以便查阅单词（键）的释义和拼写（值）。现在，由于性能良好的符号表的实现，人们期望在计算机上实现内置拼写检查器并能够立即访问计算机上的字词释义。
- **账户信息。**拥有股票的人常常会在网上查看当前价格。网络上的若干服务通常会将股票代码（键）与其当前的价格（值）关联起来，通常还包括许多其他信息（请参阅程序 3.1.8）。此类商业应用比比皆是，包括金融机构把账户信息与姓名或账号关联起来，以及教育机构将成绩与学生姓名或身份号码相关联。 [628]
- **基因组学。**符号表在现代基因组学中扮演着重要的角色。最简单的例子是使用字母 A、C、T 和 G 来表示生命体 DNA 中的核苷酸。还有一个最简单的例子是密码子（核苷酸三连体）和氨基酸之间的对应关系（TTA 对应于亮氨酸，TCT 对应丝氨酸等），以及氨基酸序列和蛋白质之间的对应关系等。基因组学的研究人员经常使用各种类型的符号表来组织这类知识。
- **实验数据。**从天体物理学到动物学，现代科学家被大量实验数据淹没，如何组织并高效地访问这些数据对于理解这些数据的含义至关重要。符号表是一个关键的起点，基于符号表的高级数据结构和算法已成为科学研究的重要组成部分。
- **编程语言。**符号表的最早应用之一是组织编程信息。起初，程序只是简单的二进制数序列，但程序员很快发现使用符号表代替操作指令和内存地址（变量名）要方便得多。将名称与数字关联需要符号表。随着程序规模的增长，符号表操作的开销成为程序开发的瓶颈，最终形成数据结构和算法的发展需求，就像我们在本节中讨论的那样。
- **文件。**我们经常使用符号表组织计算机系统的数据。也许最突出的例子就是文件系统，我们将文件名（键）与其内容所在的位置（值）关联。音乐播放器使用相同的系统将歌曲名（键）与音乐本身所在的位置（值）相关联。

- 互联网域名系统。作为在互联网上组织信息的基础的域名系统 (DNS), DNS 将人容易理解的 URL (键) (如 `www.princeton.edu` 或 `www.wikipedia.org`) 与计算机网络路由器理解的 IP 地址 (值) (如 `208.216.181.15` 或 `207.142.131.206`) 关联。这个系统是下一代“电话簿”。因而, 人类可以使用容易记忆的名字来使用网络, 同时机器可以高效地处理这些数值数据。在世界各地的互联网路由器上, 用于此目的的符号表的每秒查询数量十分巨大, 因此性能是非常重要的。每年网络都会新增数以百万计的新计算机和其他设备, 因而互联网路由器上的这些符号表必须是动态的。

尽管列举的范围很广, 上述列表还只是一些有代表性的例子, 用来向你展示符号表抽象的概念。每当你通过名称来指定某类事物时, 就有一个符号表在工作。计算机的文件系统或网络之所以能为你提供服务, 是因为其后台有一个符号表。

例如, 要构建一个将氨基酸名称与密码子相关联的符号表, 可以编写如下代码:

```
ST<String, String> amino;
amino = new ST<String, String>();
amino.put("TTA", "leucine");
...
```

将信息与关键字相关联的想法非常重要, 许多高级语言都内置了对关联数组 (associative array) 的支持, 你可以使用标准数组操作语法来进行访问, 只是方括号内是键而不是整数索引。在这种语言中, 你可以写 `amino["TTA"] = leucine` 而不用再写 `amino.put("TTA", "leucine")`。尽管 Java 还不 (尚未) 支持这种语法, 但是从关联数组的角度来看, 这是理解符号表基本用途的一个好方法。

Lookup (程序 4.4.1) 从命令行指定的一个逗号分隔值文件 (参见 3.1 节) 中构建键 - 值对集合, 然后输出与标准输入读取的键对应的值。命令行参数为文件名和两个整数, 一个整数指定作为键的字段, 另一个指定作为值的字段。

理解符号表的第一步是从本书网站下载 `Lookup.java` 和 `ST.java` (即我们在本节末尾将讨论的符号表实现), 然后执行符号表查询。你可以找到许多与我们所描述的各种应用相关的逗号分隔值 (.csv) 文件, 包括 `amino.csv` (密码子的氨基酸编码)、`DJIA.csv` (历史上每一天的股票的开盘价、交易量、收盘价和股票市场的平均值) 以及 `ip.csv` (DNS 数据库的部分项目)。在选择使用哪个字段作为键时, 请记住每个键必须唯一确定一个值。如果存在多个 `put` 操作将多个值关联到同一个键, 符号表将只记住最近的一个 (考虑关联数组)。我们将在后续章节讨论将多个值与一个键相关联的情况。

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
...
```

```
GCA,Ala,A,Alanine
GCG,Ala,A,Alanine
GAT,Asp,D,Aspartic Acid
GAC,Asp,D,Aspartic Acid
GAA,Gly,G,Glutamic Acid
GAG,Gly,G,Glutamic Acid
GGT,Gly,G,Glycine
GGC,Gly,G,Glycine
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine
```

```
% more DJIA.csv
...
20-Oct-87,1738.74,608099968,1841.01
19-Oct-87,2164.16,604300032,1738.74
16-Oct-87,2355.09,338500000,2246.73
15-Oct-87,2412.70,263200000,2355.09
...
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
...
```

```
% more ip.csv
...
www.ebay.com,66.135.192.87
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.cnn.com,64.236.16.20
www.google.com,216.239.41.99
www.nytimes.com,199.239.136.200
www.apple.com,17.112.152.32
www.slashdot.org,66.35.250.151
www.espn.com,199.181.135.201
www.weather.com,63.111.66.11
www.yahoo.com,216.109.118.65
...
```

典型的逗号分隔值 (CSV 文件)

程序4.4.1 字典查找

```
public class Lookup
{
 public static void main(String[] args)
 {
 // 创建字典, 并从标准输入读取key值
 In in = new In(args[0]);
 int keyField = Integer.parseInt(args[1]);
 int valField = Integer.parseInt(args[2]);

 String[] database = in.readAllLines();
 StdRandom.shuffle(database);

 ST<String, String> st = new ST<String, String>();
 for (int i = 0; i < database.length; i++)
 {
 // 从一行的输入信息中提取key和value, 并将其存入ST中
 String[] tokens = database[i].split(",");
 String key = tokens[keyField];
 String val = tokens[valField];
 st.put(key, val);
 }

 while (!StdIn.isEmpty())
 {
 // 读取key并输出值
 String s = StdIn.readString();
 StdOut.println(st.get(s));
 }
 }
}
```

|            |             |
|------------|-------------|
| in         | 输入流 (CSV文件) |
| keyField   | key的位置      |
| valField   | value的位置    |
| database[] | 输入的若干行      |
| st         | 符号表 (BST)   |
| tokens     | 一行输入的若干个值   |
| key        | 键           |
| val        | 值           |
| s          | 输入的查询信息     |

这个符号表客户程序从逗号分隔值 (CSV) 文件中读取键-值对, 然后打印对应于标准输入上键的值。键和值都是字符串。

```
% java Lookup amino.csv 0 3
TTA
Leucine
ABC
null
TCT
Serine
% java Lookup amino.csv 3 0
Glycine
GGG
```

```
% java Lookup ip.csv 0 1
www.google.com
216.239.41.99
% java Lookup ip.csv 1 0
216.239.41.99
www.google.com
% java Lookup DJIA.csv 0 1
29-Oct-29
252.38
```

在本节稍后的部分, 我们将了解到 Lookup 程序中 put 操作和 get 请求的时间开销与符号表大小的对数成正比。这个事实意味着针对第一次 get 请求, 你可能会经历一个小的延迟 (对于所有 put 操作都会有这个延迟), 但其后的请求响应速度很快。

索引。Index (程序 4.4.2) 是一个典型的符号表客户程序, 它交替地调用了一系列 get() 和 put(): 程序从标准输入中读取一个字符串序列, 并打印出一个排好序的字符串表, 每个字符串对应一个整数列表, 用于指示字符串在输入中的位置。我们有大量的数据, 并希望知道某些感兴趣的字符串在哪里出现。在这种情况下, 我们似乎是将多个值与一个键相关联 (因为一个词可能多次出现——译者注), 但实际上只关联了一个值: 因为我们把这些值存入了一个队列。Index 使用两个整数型命令行参数来控制输出: 第一个整数是符号表中最小字符串的长度, 第二个是被打印的字符串的最少出现次数。以下列出了 Index 程序在不同领域中的应用。

- 图书索引。每本教科书都有一个索引, 可以用于查找一个单词并获取包含该单词的页码。虽然没有读者希望书中的每一个单词都包括在索引中, 类似 Index 这样的程序可以帮助我们创建一个优质的索引 (因为 Index 程序可以选择将出现频度较高的词列出来——译者注)。

程序4.4.2 索引

```
public class Index
{
 public static void main(String[] args)
 {
 int minlen = Integer.parseInt(args[0]);
 int minocc = Integer.parseInt(args[1]);
 // 创建并初始化符号表
 ST<String, Queue<Integer>> st;
 st = new ST<String, Queue<Integer>>();
 for (int i = 0; !StdIn.isEmpty(); i++)
 {
 String word = StdIn.readString();
 if (word.length() < minlen) continue;
 if (!st.contains(word))
 st.put(word, new Queue<Integer>());
 Queue<Integer> queue = st.get(word);
 queue.enqueue(i);
 }
 // 输出出现次数超过阈值的单词
 for (String s : st)
 {
 Queue<Integer> queue = st.get(s);
 if (queue.size() >= minocc)
 StdOut.println(s + ": " + queue);
 }
 }
}
```

|        |           |
|--------|-----------|
| minlen | 最短长度      |
| minocc | 计数阈值      |
| st     | 符号表       |
| word   | 当前单词      |
| queue  | 当前单词的位置队列 |

这个符号表的客户程序为一个文本文件建立一个索引，键是单词，值是文件中单词出现的位置的队列

```
% java Index 9 30 < TaleOfTwoCities.txt
confidence: 2794 23064 25031 34249 47907 48268 48577 ...
courtyard: 11885 12062 17303 17451 32404 32522 38663 ...
evremonde: 86211 90791 90798 90802 90814 90822 90856 ...
...
something: 3406 3765 9283 13234 13239 15245 20257 ...
sometimes: 4514 4530 4548 6082 20731 33883 34239 ...
vengeance: 56041 63943 67705 79351 79941 79945 80225 ...
```

633

- 编程语言。在大型程序中往往会使用大量的符号（变量名或函数名），了解每个名称的使用位置是非常有必要的。类似 Index 的程序可以发挥非常重要的作用，帮助程序员跟踪程序中符号的使用位置。历史上，打印出一个符号表是程序员用来管理大型程序的最重要的工具之一。在现代编程语言中，符号表是编程人员用于管理系统中模块名称的软件工具的基础。

|      | 键     | 值       |
|------|-------|---------|
| 书籍索引 | 单词    | 页码      |
| 基因组学 | DNA序列 | 基因组中的位置 |
| 网络搜索 | 关键词   | 网站列表    |
| 业务信息 | 客户姓名  | 交易信息    |

典型的索引应用程序

- 基因组学。在一个典型的（可能有点过于简化）基因组学研究场景中，一个科学家往往希望知道一个给定的基因序列在一个现有的基因组或基因组集中的位置。某些序列是

否存在或者与哪些序列邻近，可能都具有重要的科学意义。这类科学研究的起点是类似于程序 Index 产生的索引，需要做的修改是，要考虑到基因组不会被分割成“单词”。

- 网络搜索。当你键入一个关键字，获取包含该关键字的网站列表时，你就在使用 Web 搜索引擎创建的索引，在这个索引中，值是网页的列表，键是查询的单词。当然实际情况会更加动态和复杂，因为我们经常指定多个键，并且页面遍布整个网络，而不是保存在单个计算机中。
- 账户信息。如果一个公司想为客户账户记录客户一天的交易信息，一种方法就是保存交易列表的索引。键为账号；值是在交易列表中该账号出现的列表。

我们特别希望你能从本书官网下载程序 Index，并使用不同的输入文件运行程序，以便进一步了解符号表的应用。如果你这样做，你会发现这个程序可以在短时间内为大型文件构建大型索引，而且每个 put 操作和 get 请求都能立即响应。为大型的符号表提供快速响应是算法技术的经典贡献之一。

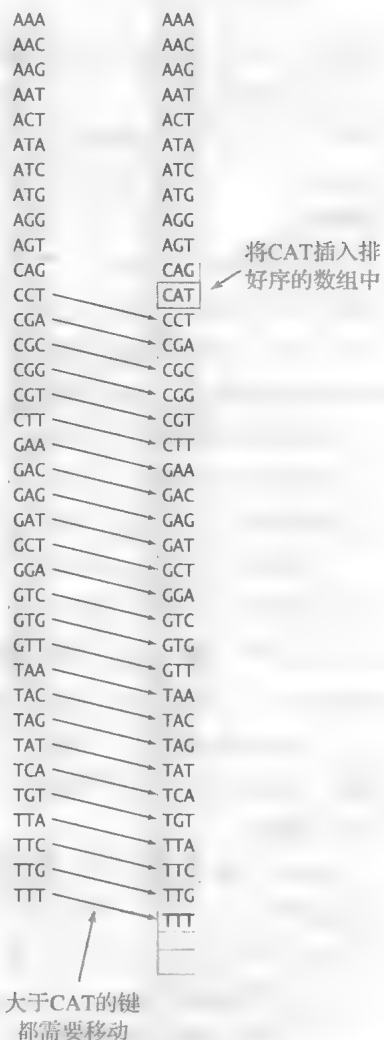
634

**基本的符号表实现** 上述所有的例子都是符号表重要性的有力证据。符号表的实现已经被深入研究，为此设计了许多不同的算法和数据结构。现代编程环境（如 Java）往往会包括一个（或更多）符号表实现。按照惯例，了解一个最基础的符号表的工作原理将帮助你理解那些高级的实现，并能更好地选择和更有效地使用它们，同时也能帮助你在必要时设计自己的符号表实现（用于解决一些特定场景的专用问题）。

首先，我们简单讨论两个基本实现，基于我们使用过的两种基本数据结构：可变数组和链表。我们这样做的目的是让你明白：我们需要一个更为复杂的数据结构，因为每个实现的 put 操作和 get 操作都需要线性运行时间，这使得它们都不适合大型的实用程序。

也许最简单的实现是将键-值对存储在一个无序的链表（或数组）中，使用顺序搜索（sequential search）（参见练习 4.4.6）来查找数据。顺序搜索意味着在搜索键时，会依次检查每个节点（或元素），直到找到指定的键或遍历完整个链表（或数组）。这种实现方式对于典型的客户程序来说是不可行的，因为当键不在符号表中时 get 操作花费了线性时间。

或者，我们可以使用键的排序（可变）数组和值的平行数组（平行数组是指键和值各占一个数组，配对的键和值的下标是一致的——译者注）。由于键是已经排好序的，因此我们可以使用二分查找实现键（及关联值）的搜索，如 4.2 节所述。基于二分查找法实现一个符号表并不困难（具体参见练习 4.4.5）。这种实现中，查找是快速的（对数运行时间），但是插入通常很慢（线性运行时间），因为必须按照键的排序顺序维护一个可变数组。每次插入一个新的键时，所有大的键必须在数组中后移一个位

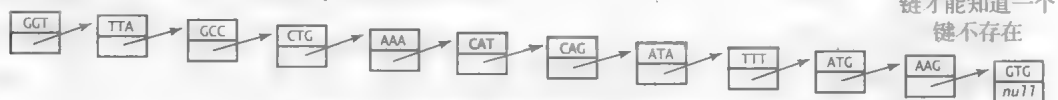


在一个有序的数组中插入数据需要线性时间



置，这意味着在最坏的情况下 put 操作需要线性运行时间。

链表（无序）



链表中的顺序搜索需要线性运行时间

为了实现一个适用于 Lookup 和 Index 的符号表，我们需要一个比链表或可变数组更灵活的数据结构。接下来，我们将讨论两种这样的数据结构：散列表和二叉搜索树。

**散列表** 散列表是一种数据结构，它把键分成小的组，可以实现快速查找。我们选择一个参数  $m$ ，把所有键分成  $m$  组，期望组的大小大致相同。在每个组内部，我们使用未排序的列表保存键并使用顺序查找算法，就像前文讨论的基本实现方法。

为了实现把键划分为  $m$  个组，我们使用一个称为散列函数（hash function）的函数。散列函数将每个键映射为一个散列值——一个位于 0 到  $m-1$  之间的整数。这使得我们可以把符号表建模为一个长度固定的数组，其元素为链表，并使用散列值作为数组索引下标来访问所需的链表。

散列算法非常有用，所以很多编程语言都提供了对它的直接支持。正如 3.3 节所述，每个 Java 类都应该有一个 hashCode() 方法正是为了用于此目的。如果你使用的是非标准类型，则需要检查 hashCode() 是否被正确实现，因为默认情况下可能无法将键划分为相同大小的组。为了将散列码转换为 0 到  $m-1$  之间的散列值，我们使用表达式  $\text{Math.abs}(x.\text{hashCode}() \% m)$ 。

只要两个对象相等（通过 equals() 方法判断），它们就具有相同的散列码。但是，不相等的对象其散列码也可能相同。最后设计散列函数时，应该尽量使得调用  $\text{Math.abs}(x.\text{hashCode}() \% m)$  时以相等的概率从 0 到  $m-1$  之间取值。

上面的表格是 12 个具有代表性的字符串键的散列码和散列值，其中  $m=5$ 。注意：一般来说，散列码为在  $-2^{31}$  和  $2^{31}-1$  之间的整数，对于较短的字母数字字符串，散列码也会是较小的正整数。

| 键   | 散列码   | 散列值 |
|-----|-------|-----|
| GGT | 70516 | 1   |
| TTA | 83393 | 3   |
| GCC | 70375 | 0   |
| CTG | 67062 | 2   |
| AAA | 64545 | 0   |
| CAT | 66486 | 1   |
| CAG | 66473 | 2   |
| ATA | 65134 | 4   |
| TTT | 83412 | 2   |
| ATG | 65140 | 0   |
| AAG | 64551 | 1   |
| GTG | 70906 | 1   |

12 个字符串键 ( $n=12, m=5$ ) 的散列码和散列值

通过上述准备工作，使用散列算法直接扩展 4.3 节中讨论的链表代码，就可以实现一个高效的符号表。我们维护一个由  $m$  个链表组成的数组，其中数组元素  $i$  是一个包含所有散列值为  $i$  的键（连同其关联值）的链表。当定位一个键时：

- 计算其散列值以定位其链表。
- 迭代该链表中的节点，查找该键。
- 如果定位键在链表中，则返回键的关联值；否则，返回 null。

当想要插入一个键-值对时：

- 计算键的散列值以定位其链表。
- 迭代该链表中的节点，查找该键。

- 如果键在链表中，则将当前与键关联的值替换为新值；否则，使用给定的键和值创建一个新节点，并将其插入链表的开头。

HashSet（程序 4.4.3）是一个完整的实现，使用固定数量（ $m=1024$ ）的链表。它依赖于以下嵌套类来表示链表中的每个节点：

```
private static class Node
{
 private Object key;
 private Object val;
 private Node next;

 public Node(Object key, Object val, Node next)
 {
 this.key = key;
 this.val = val;
 this.next = next;
 }
}
```

程序 HashST 的效率取决于  $m$  的值和散列函数的好坏。假设散列函数可以合理地分配键，则程序性能比链表中的顺序查找快约  $m$  倍，其代价是  $m$  个额外的引用和链表。这是一个经典的时空折中方案： $m$  的值越大，使用的内存就越多，但消耗的时间越少。

637

程序4.4.3 散列表

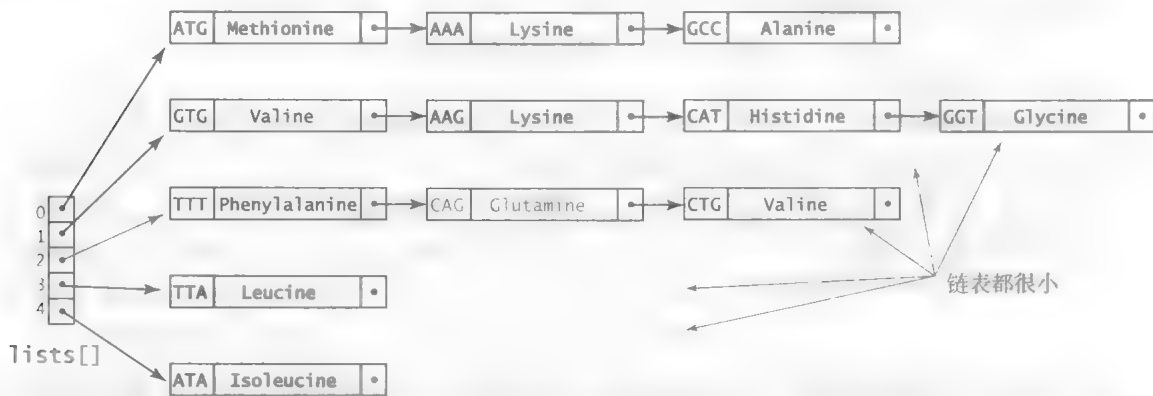
```
public class HashST<Key, Value>
{
 private int m = 1024;
 private Node[] lists = new Node[m];
 private class Node
 {
 /* 见对应正文内容 */
 }
 private int hash(Key key)
 { return Math.abs(key.hashCode() % m); }
 public Value get(Key key)
 {
 int i = hash(key);
 for (Node x = lists[i]; x != null; x = x.next)
 if (key.equals(x.key))
 return (Value) x.val;
 return null;
 }
 public void put(Key key, Value val)
 {
 int i = hash(key);
 for (Node x = lists[i]; x != null; x = x.next)
 {
 if (key.equals(x.key))
 {
 x.val = val;
 return;
 }
 }
 lists[i] = new Node(key, val, lists[i]);
 }
}
```

|          |         |
|----------|---------|
| $m$      | 链表的数量   |
| lists[i] | 散列值i的链表 |

这个程序使用一个链表来实现散列表。散列函数选择  $m$  个链表中的一个。如果表中存在  $n$  个键，则对于合适的 `hashCode()` 实现，`put()` 或 `get()` 操作的平均代价为  $n/m$ 。如果我们使用一个可变数组来保证每个链表中键的平均数量在 1 和 8 之间（参见练习 4.4.12），则每次操作的运行时间为常量。我们把 `contains()`、`keys()`、`size()` 和 `remove()` 留到练习 4.4.8 ~ 4.4.11 来实现。

638

下图显示了为示例中的键所构建的散列表（按照前文表格所给的顺序插入）。首先，GGT 被插入链表 1，然后 TTA 被插入链表 3，然后 GCC 被插入链表 0 中，以此类推。散列表构建完之后开始查找 CAG，通过计算 CAG 的散列值（计算结果为 2），然后顺序地查找链表 2。在找到链表 2 的第二个节点中的键 CAG 之后，方法 `get()` 返回值 `Glutamine`。



$m=5$  时的散列表

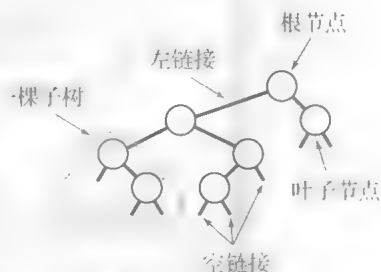
通常情况下，程序员通过大致估计要处理的键的数量，来选择 一个大的固定值  $m$ （例如，默认情况下我们选择 1024）。通过使用可变数组 `lists[]`，我们可以确保每个链表的平均键数是常量。例如，练习 4.4.12 展示了如何确保每个链表的平均键数在 1 和 8 之间，从而使得 `put` 和 `get` 操作的运行性能均为常量。肯定需要调整这些参数来更好地适应实际应用场景。

散列表的主要优点是它们可以高效地完成 `put` 和 `get` 操作。散列表的一个缺点是它们没有利用键的顺序，从而不能保证键按顺序排列（或无法支持基于排序的其他操作）。例如，如果在 `Index` 中用 `HashST` 替换 `ST`，那么这些键可能会以杂乱的顺序输出，而不是按照顺序。或者，如果想找到最小的键或最大的键，我们必须查找全部的键。接下来，我们讨论另一个符号表实现，在键可比较的情况下支持基于顺序的操作，而无须牺牲 `put()` 和 `get()` 的性能。

**二叉搜索树** 二叉树是一种数学抽象，在高效的信息组织中扮演重要的角色。二叉树是一个递归定义的结构，可以是如下类型：空树（`null`），或者是一个节点包含着到两个不相交的二叉树的链接。二叉树在计算机编程中起着重要的作用，因为它提供了实现的灵活性和方便性之间的有效平衡。二叉树在科学、数学和计算方面有很多应用，你肯定会在很多场合遇到这个模型。

讨论二叉树时，我们通常使用基于树的术语。我们把位于顶部的节点称为树的根，左链接引用的节点称为左子树，右链接引用的节点称为右子树。按传统，计算机科学家将树倒置，顶端是根。左右链接均为 `null` 的节点称为叶子节点。树的高度是从根节点到叶节点的所有路径中节点最多的那条路径的链接数。

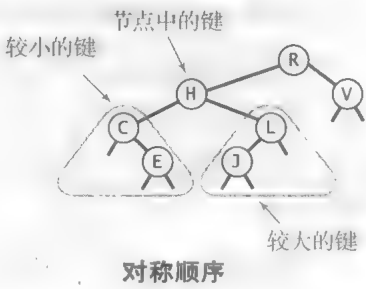
与数组、链表和散列表一样，我们使用二叉树来存储数据的集合。对于符号表的实现，我们使用一种称为二



一棵二叉树的剖析图

搜索树 (BST) 的特殊类型的二叉树。二叉搜索树是一个二叉树，每个节点包含一个键值对，并且键有如下对称顺序：每个节点的键大于其左子树中每个节点的键，并小于其右子树中每个节点的键。下面你将看到，对称排序能够高效地实现 put 和 get 操作。

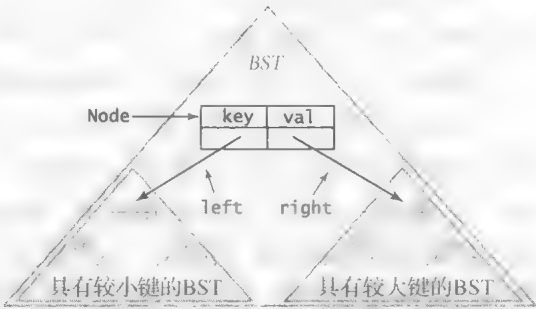
为了实现二叉搜索树，我们从节点抽象的类开始，它包括四个引用，分别指向键、值、左二叉搜索树和右二叉搜索树。键类型必须实现 Comparable(用于指定键的顺序)，但值类型是任意的。



640

这个定义类似于我们对链表节点的定义，除了它包含两个链接，而不是一个链接。与链表一样，递归数据结构的思想可能有些费解，但其实我们所做的只是在链表的定义中添加了第二个链接（并加入排序）。

为了（稍微）简化代码，我们在 Node 中添加了一个构造函数，用于初始化 key 和 val 实例变量：



二叉搜索树的示意图

```
Node(Key key, Value val)
{
 this.key = key;
 this.val = val;
}
```

new Node(key, val) 的结果是一个指向 Node 对象的引用（我们可以将其赋值给 Node 类型的任何变量），其 key 和 val 实例变量被设置为给定的值，其 left 和 right 实例变量均初始化为 null。

与链表一样，当跟踪使用二叉搜索树的代码时，我们可以使用一种可视化表示方式：

- 绘制一个矩形框来表示每个对象。
- 将实例变量放在矩形框中。
- 绘制箭头指向引用的对象。

在大多数情况下，我们使用一种更简单的抽象表示，绘制包含键的矩形框（或圆形）表示节点（省略值），使用箭头连接节点以表示链接。这个抽象表示使我们能够主要关注链接结构。

下面给出一个例子，我们考虑一个带有字符串键和整数值的二叉搜索树。要构建一个将 0 与键 “it” 关联的单节点的二叉搜索树，我们创建一个 Node：

```
Node first = new Node("it", 0);
```

由于左右链接均为 null，所以该节点表示含有一个节点的二叉搜索树。要添加将值 1 与键相关联的节点，我们创建另一个节点：

```
Node second = new Node("was", 1);
```

641

(这本身就是一个二叉搜索树)，并将第一个 Node 的右字段链接到此节点：

```
first.right = second;
```

第二个节点必须位于第一个节点的右侧，因为“it”比“was”的字母顺序小（或者，我们可以设置 `second.left` 为 `first`）。现在我们可以添加第三个节点，把键“the”与值 2 关联：

```
Node third = new Node("the", 2);
second.left = third;
```

再使用如下代码把第四个键“best”与值 3 相关联：

```
Node fourth = new Node("best", 3);
first.left = fourth;
```

请注意，根据定义，每个链接（`first`、`second`、`third` 和 `fourth`）都是二叉搜索树（每个链接或者为 `null` 或者引用二叉搜索树，并且在每个节点上都满足排序条件）。

在目前的情况下，注意确保总是将节点连接在一起，并且能够使得我们创建的每个节点都是二叉搜索树的根（有一个键，一个值，左侧链接到一个较小值的二叉搜索树，右侧链接到一个较大值的二叉搜索树）。从二叉搜索树数据结构的角度来看，值并不重要，所以我们在图中经常忽略它，但是我们把它包含在定义中，因为它在符号表概念中扮演着重要的角色。我们稍微滥用了命名法，使用 `ST` 同时来表示“符号表”和“搜索树”，因为搜索树在符号表实现中扮演着如此重要的角色。

一个二叉搜索树代表了一个有序的项序列。在刚刚讨论的例子中，`first` 代表序列“best it the was”。我们也可以使用数组来表示一个有序序列。例如，我们可以使用

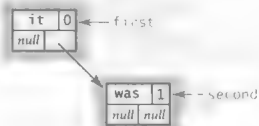
642 `String[] a = { "best", "it", "the", "was" };`

表示与上述相同的有序字符串序列。给定一个不同键的集合，存在唯一的方法把集合表示为一个有序数组，但是存在许多方法把集合表示成一个二叉搜索树（具体参见练习 4.4.7）。这种灵活性允许我们开发高效的符号表实现。例如，在我们的例子中，通过创建一个新节点并且仅修改一个链接就可以插入新的键值对。可以证明，只改动一个链接就可以处理所有的情况。同样重要的是，我们可以很容易地在树中找到一个指定的键所在的位置，也可以找到用于增加一个新键所需要改动的链接的位置。接下来，我们讨论实现上述任务的符号表代码。

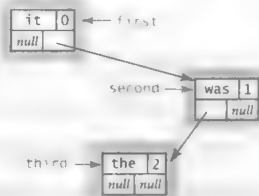
```
Node first = new Node("it", 0);
```



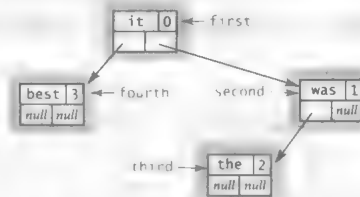
```
Node second = new Node("was", 1);
first.right = second;
```



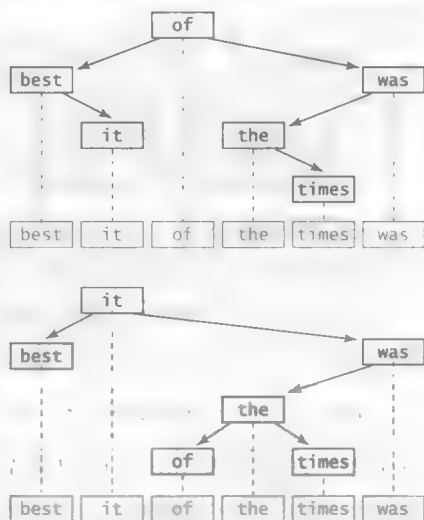
```
Node third = new Node("the", 2);
second.left = third;
```



```
Node fourth = new Node("best", 2);
first.left = fourth;
```

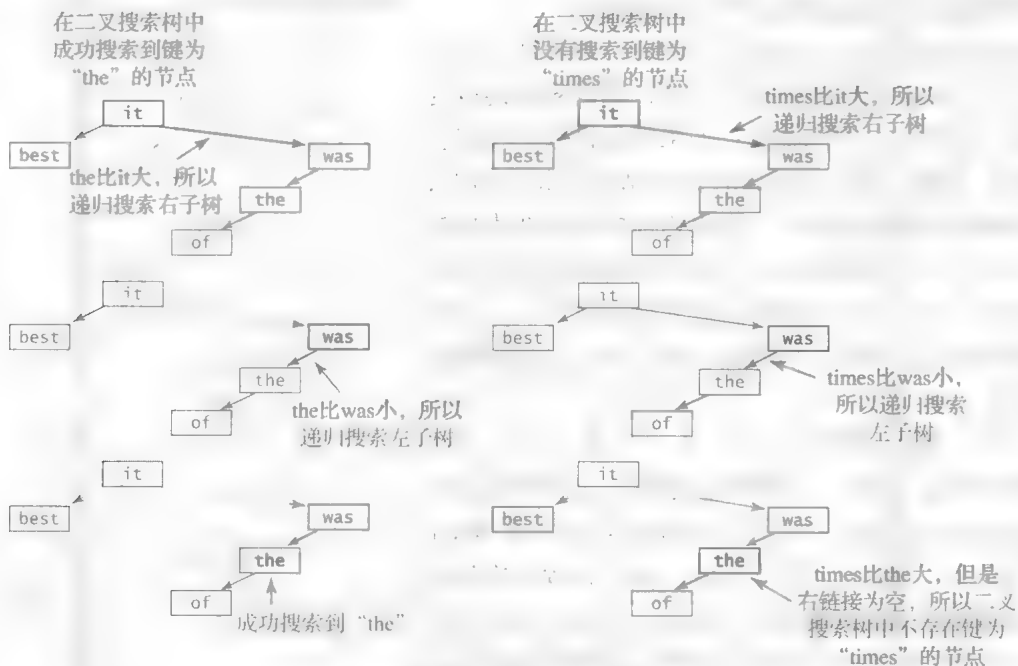


将二叉搜索树连接在一起



表示同一序列的两个不同的二叉搜索树

搜索。假设需要在二叉搜索树中查找具有给定键的节点（或者在一个符号表中调用 get 获取给定键的值）。存在两种可能的结果：搜索成功（在二叉搜索树中找到键；在符号表实现中，我们返回关联的值）或者搜索失败（二叉搜索树不存在给定的键；在符号表实现中我们返回 null）。



二叉搜索树搜索示意图

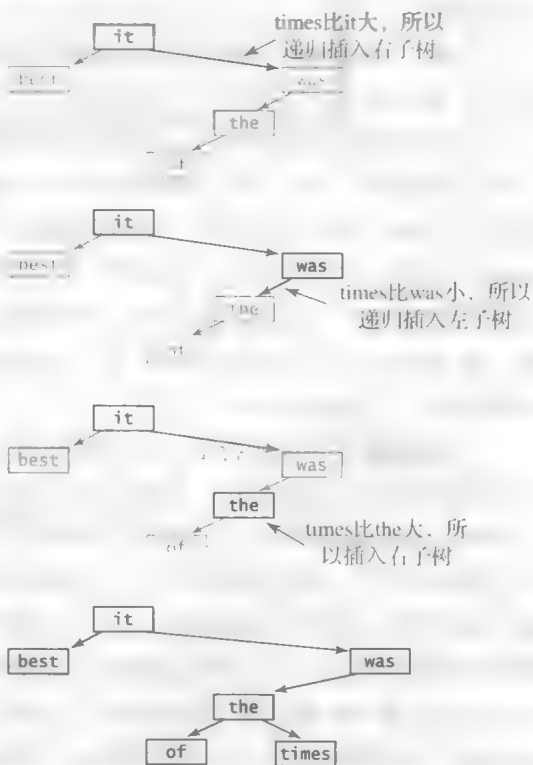
643

显而易见比较合理的实现方式是采用递归搜索算法：给定一个二叉搜索树（指向一个 Node 的引用），首先检查树是否为空（即引用是否为 null），如果树为空，则结束搜索，提示不成功（在符号表实现中返回 null）；如果树不为空，检查节点中的键是否等于搜索的键，如果相等，则结束搜索，提示成功（在符号表实现中返回与该键相关联的值）；如果不是，则将搜索的键与节点中的键进行比较，如果较小，则在左子树中搜索（递归）；如果较大，则在右子树中搜索（递归）。

从递归角度看，证明这个方法的可行性和正确性并不困难，基于“如果在二叉搜索树中存在，当且仅当该键在当前子树中”的不变原则。递归方法的关键特性是我们永远只须检查一个节点以确定下一步采取何种行动。此外，我们通常仅仅检查树中的少部分节点：因为无论何时我们进入一个节点的子树，我们就永远不会检查该节点另一个子树中的节点。

插入。假设我们需要插入一个新的节点到

插入“times”



将一个新节点插入二叉搜索树

一个二叉搜索树中（在符号表实现中，对应于使用 `put` 操作把一个新的键-值对放入数据结构中），其逻辑类似于搜索一个键，但是实现稍微复杂。理解插入操作的关键是要认识到仅仅需要修改一个链接以指向新节点，而这个链接恰恰是搜索该键失败时查找结果为 `null` 的链接。

如果树是空的，则创建并返回一个包含键-值对的新节点；如果搜索键小于根的键，则设置左链接为插入键-值对到左树的结果；如果搜索键较大，则设置右链接为插入键-值对到右树的结果；其他情况，如果搜索键与根的键相等，则将当前的值更新为新的值。以这种方式递归调用后，重置左链接或右链接是不必要的，因为只有子树为空时才会修改链接，虽然重置链接很容易做到，但我们会通过检查以尽量避免这个操作。

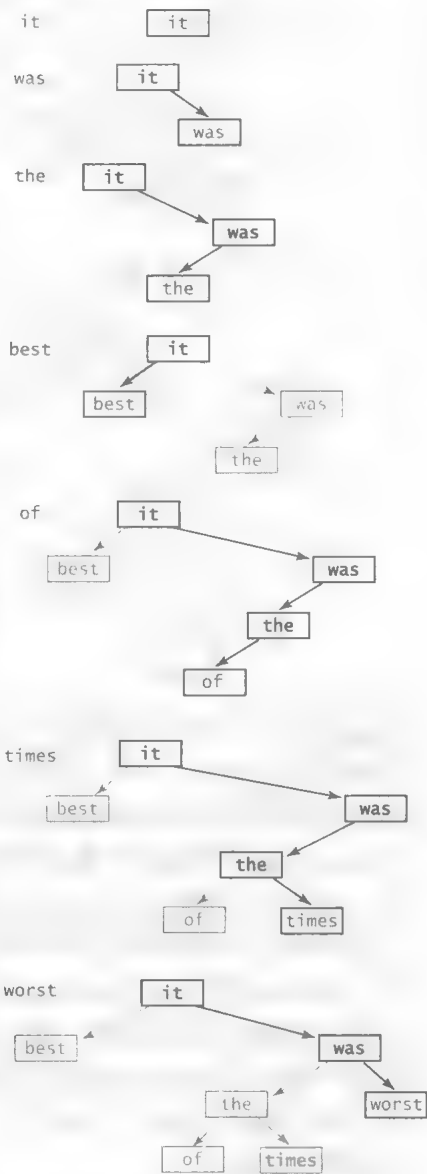
实现。BST（程序 4.4.4）是一个基于两个递归算法的符号表实现。如果将此代码与二分查找实现 `BinarySearch`（程序 4.2.3）以及栈和队列实现 `Stack`（程序 4.3.4）、`Queue`（程序 4.3.6）进行比较，你将会发现此代码更加优雅和易读。请花些时间仔细思考递归，以确保你能正确理解这段代码。也许最简单的方法是跟踪并观察根据一系列样本键构造一棵二叉搜索树的过程。这可以算是一个对基本数据结构的理解程度的测试。

此外，BST 中的 `put()` 和 `get()` 方法效率非常高：通常，每个方法仅仅访问二叉搜索树中的少量节点（从根节点到搜索到的节点，或者替换为连接到新节点的 `null` 链接）。接下来，我们将证明 `put` 操作和 `get` 请求仅会消耗对数运行时间（在某些假设下）。另外，`put()` 只创建一个新的节点并添加一个新的链接。如果你通过插入一些键到初始的空树来构建二叉搜索树，你肯定会确信这个事实——因为每次都只是在树底部绘制一个新的节点。

**二叉搜索树的性能特征** 二叉搜索树算法的运行时间最终取决于树的形状，而树的形状取决于键的插入顺序。理解这种依赖关系是在实际情况中有效使用二叉搜索树的关键因素。

**最佳情况。**对于最佳情况，二叉搜索树是一棵完全平衡的树（每个节点恰好包含两个非空的子节点），根节点到叶子节点的节点数为  $\lg n$ 。在这样的树中，很容易发现一种失败搜索的代价为对数型，因为其代价满足与二分查找算法相同的递归关系式（参见 4.2 节），因而每个 `put` 操作和 `get` 请求的代价正比于  $\lg n$  或者更少。这在实际应用中是很少见的，需要非常好的运气才能通过逐一插入键获得一棵完全平衡的树，但还是有必要了解 BST 在这种情况下的最佳性能特点。

要插入二叉  
搜索树中的键



构建一个 BST



程序4.4.4 二叉搜索树

```
public class BST<Key extends Comparable<Key>, Value>
{
 private Node root;

 private class Node
 {
 Key key;
 Value val;
 Node left, right;
 Node(Key key, Value val)
 { this.key = key; this.val = val; }
 }

 public Value get(Key key)
 { return get(root, key); }

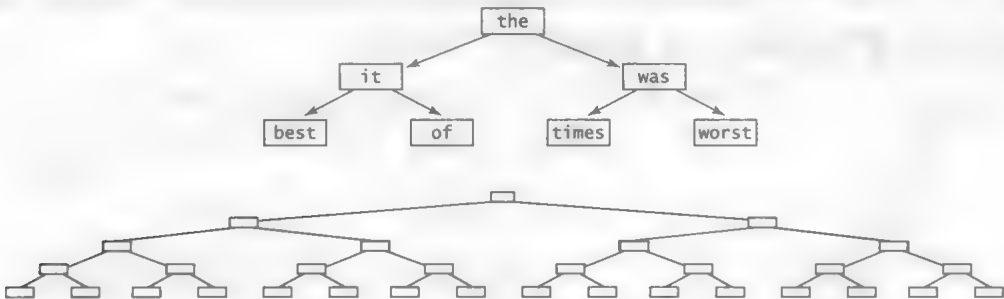
 private Value get(Node x, Key key)
 {
 if (x == null) return null;
 int cmp = key.compareTo(x.key);
 if (cmp < 0) return get(x.left, key);
 else if (cmp > 0) return get(x.right, key);
 else return x.val;
 }

 public void put(Key key, Value val)
 { root = put(root, key, val); }

 private Node put(Node x, Key key, Value val)
 {
 if (x == null) return new Node(key, val);
 int cmp = key.compareTo(x.key);
 if (cmp < 0) x.left = put(x.left, key, val);
 else if (cmp > 0) x.right = put(x.right, key, val);
 else x.val = val;
 return x;
 }
}
```

|       |       |
|-------|-------|
| root  | BST的根 |
| key   | 键     |
| val   | 值     |
| left  | 左子树   |
| right | 右子树   |

以递归BST数据结构为中心的符号表数据类型的实现，使用递归的方法实现遍历。我们将contains()、size()和remove()的实现留到练习4.4.18 ~ 4.4.20。我们在本节结尾处实现了keys()。

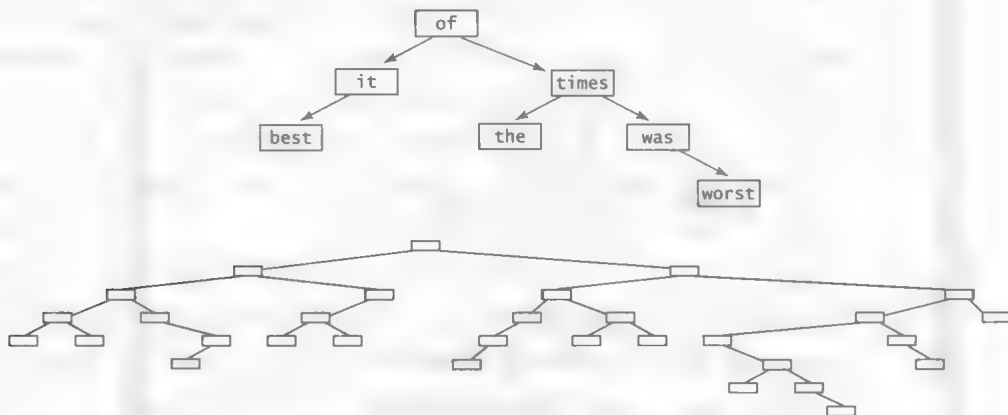


最佳情况下的二叉搜索树（完全平衡树）

646  
~  
647

一般情况。如果按照随机顺序插入键，我们可以期望搜索时间也是对数型，因为第一个键成为树的根，其他键会大致分成两部分。对于子树也会有同样的规律，我们期望获得与最佳情况相同的结果。事实上，这种直觉可以通过仔细分析得到证明：一个经典的数学推导表明，在根据随机顺序键构造的树中，其 put 和 get 操作所需的时间是对数型（请参阅本书网站上的参考资料）。更确切地说，在根据  $n$  的随机顺序键构造的树中，随机的 put 和 get 操作

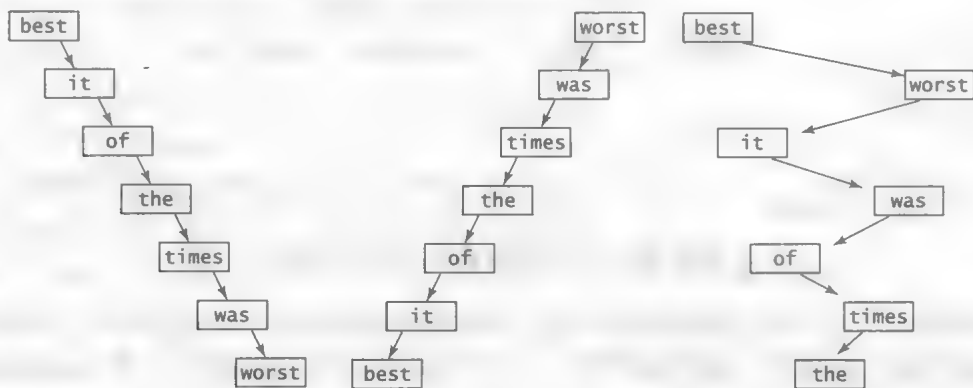
期望的键比较次数为  $\sim 2\ln n$ 。在实际应用程序中，例如程序 Lookup，当我们可以控制键的顺序时，我们会刻意地使键值随机、以大概率保证结果的对数型性能。事实上，因为  $2\ln n$  约为  $1.39\ln n$ ，所以一般情况只比最佳情况大约多出 39%。在 Index 这样的应用程序中，我们无法控制插入的顺序，因此不能保证性能，但典型的数据仍然可以达到对数性能（具体参见练习 4.4.26）。与二分查找一样，这个事实十分重要，因为对数级和线性算法性能差异巨大：使用基于二叉搜索树的符号表实现，即使针对巨大的符号表，我们也可以实现每秒数百万次操作（或更多）。



根据随机顺序键构造出的典型二叉搜索树

**最坏情况。**在最坏情况下，每个节点（除了一个节点）总是会包含一个空链接，因而二叉搜索树与链表相似，附带一个无用的链接，其中 put 操作和 get 请求需要线性时间。不幸的是，这种最糟糕的情况在实践中并不少见。例如，当按顺序插入键时就会产生这种情况。

因此，基本二叉搜索树实现的良好性能取决于键是否随机，从而决定树是不是会包含许多长的路径。如果无法保证这个假设是合理的，则不要使用简单的二叉搜索树。这意味着，很可能当问题规模增加时相应时间减慢（注意：在软件中遇到这种情况并不罕见！）。令人欣喜的是，存在其他二叉搜索树的变种可以消除最坏情况，并通过构造近乎完全平衡的树以保证每次操作的对数时间性能。一个流行的变种称为红黑树。



最坏情况下的二叉搜索树

**遍历二叉搜索树** 树遍历（tree traversal）也许是树的最基本的处理功能：给定一棵树（引用），我们希望系统地处理树中的每一个节点。对于链表，我们通过跟踪从一个节点到另

一个节点的单个链接来完成这个任务。但是对于二叉树，有两个链接可以选择。递归很显然是一种解决方案。为了处理二叉搜索树中的所有键，我们执行如下步骤：

- 处理左子树中的每个节点。
- 处理根节点。
- 处理右子树中的每个节点。

这种方法称为树的中序遍历 (inorder tree traversal)，以区分于前序遍历 (先处理根节点) 和后序遍历 (最后处理根节点)。这些树遍历的算法经常出现在各类应用中。给定一个二叉搜索树，使用数学归纳法可以很容易证明这种方法不仅会处理二叉搜索树中的每一个键，而且会按照键的顺序依次处理。例如，如下方法按照从小到大的顺序输出参数指定的二叉搜索树中的所有键：

```
private void traverse(Node x)
{
 if (x == null) return;
 traverse(x.left);
 StdOut.println(x.key);
 traverse(x.right);
}
```

首先，按照键的顺序依次输出左子树中的所有键。接着输出根节点。然后按照键的顺序输出右子树中的所有键。

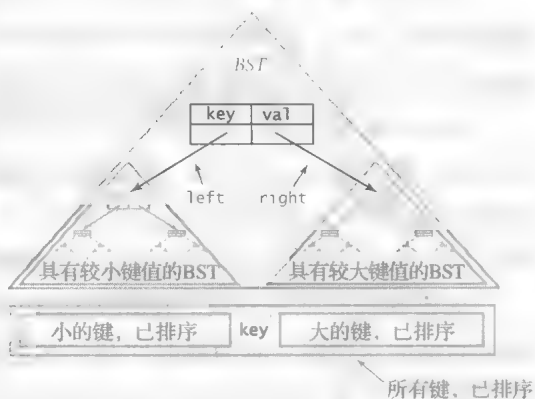
这种非常简单的方法值得仔细研究。该方法可以作为二叉搜索树的 `toString()` 实现的基础 (具体参见练习 4.4.21)，也可以作为 `keys()` 方法实现的基础，允许客户程序使用 `Java foreach` 按照排序顺序循环遍历二叉搜索树中的键 (回顾一下，这个功能在散列表中不可用，因为没有顺序)。接下来我们讨论这个遍历的基本应用。

**迭代键。**仔细看一下刚刚讨论的递归 `traverse()` 方法，它提供了一种逐个处理二叉搜索树数据类型中所有键-值对的方法。简单起见，我们只迭代键，因为我们可以在需要的时候获得键关联的值。我们的目标是实现一个方法 `keys()`，使得客户程序可以按照下面的方法使用：

```
BST<String, Double> st = new BST<String, Double>();
...
for (String key : st.keys())
 StdOut.println(key + " " + st.get(key));
...
```

`Index` (程序 4.4.2) 是客户程序代码的另一个例子，它使用 `foreach` 循环遍历键-值对。实现 `keys()` 最简单的方案是将可迭代集合中的所有键收集到一个集合中 (如栈或队列)，然后将可迭代的结果返回给客户程序。

```
public Iterable<Key> keys()
{
 Queue<Key> queue = new Queue<Key>();
 inorder(root, queue);
 return queue;
}
```



二叉搜索树的中序递归遍历

```

 }

 private void inorder(Node x, Queue<Key> queue)
 {
 if (x == null) return;
 inorder(x.left, queue);
 queue.enqueue(x.key);
 inorder(x.right, queue);
 }

```

当你第一次看到树遍历的代码，一定会觉得十分神奇。有序迭代在本质上是为了快速搜索和快速插入而设计的数据结构。请注意，我们可以使用类似的技术（即在一个可迭代集合中收集键）来实现 HashST 的 keys() 方法（具体参阅练习 4.4.10）。然而，这种实现的键将以任意顺序出现，因为散列表是无序的。

650

**有序符号表的操作** 二叉搜索树的灵活性和键可比较的能力，使得我们能比散列表支持更多有用的操作。基于二叉搜索树开发了许多其他重要的操作，这些操作在实际中有广泛的应用，下面的清单列出了有代表性的一部分。我们把这些操作的实现留作练习，并将进一步的性能特征研究和应用留给算法和数据结构课程。

**最小值和最大值。**为了获得一个二叉搜索树中的最小键，可以从根节点开始，跟随左链接直至到达 null。遇到的最后一个关键字是二叉搜索树中最小的。按照同样的过程，沿着右链接就可以找到二叉搜索树中最大的关键字（见练习 4.4.27）。

**大小和子树大小。**为了跟踪一个二叉搜索树的节点数量，可以在二叉搜索树中保存一个额外的实例变量 n，用于保存树的节点数量。把 n 初始化为 0，每当创建一个新的节点，把 n 增加 1。另外，在每个节点中再保存一个额外的实例变量 n，用于计算以该节点为根节点的子树中节点的数量（具体参见练习 4.4.29）。

**范围查找和范围计数。**使用类似于 inorder() 的递归方法，可以返回一个位于两个给定值之间键的迭代器，其消耗时间与二叉搜索树的高度加上范围中键的数量成正比（具体参见练习 4.4.31）。如果我们在每个节点中保存一个实例变量以存储各节点为根的子树的大小，我们就可以统计位于两个给定值之间键的数量，其消耗时间正比于二叉搜索树的高度（具体见练习 4.4.31）。

**层序统计和排位。**如果我们在每个节点中保存一个实例变量来保存各节点为根的子树的大小，我们可以实现一个递归方法来返回第 k 个最小键，其消耗时间与二叉搜索树的高度成正比（具体参见练习 4.4.55）。同样，我们可以计算一个键的排位，以表示二叉搜索树中严格小于该键的键的数量（见练习 4.4.56）。

今后，我们将使用 Java 的 java.util.TreeMap 来完成我们的有序符号表 API 的参考实现 ST，这是一种基于红黑树的符号表实现。如果你还选修了数据结构和算法等高级课程，你会学习到更多关于红黑树的知识。它们支持 get()、put() 以及刚刚描述的许多其他操作，并能够提供对数运行时间保证。

651

**集合数据类型** 作为最后一个例子，我们讨论一种比符号表更简单的数据类型，其应用也十分广泛，并且使用散列表或二叉搜索树可以很容易地实现。一个 set 是不同键的集合，类似于符号表但不包含值。我们可以使用 ST 并忽略这些值，但使用以下 API 的客户程序代码会更简洁、更清晰：

|                                               |           |
|-----------------------------------------------|-----------|
| public class SET<Key extends Comparable<Key>> |           |
| SET()                                         | 新建一个空的集合  |
| boolean isEmpty()                             | 集合是否为空?   |
| void add(Key key)                             | 向集合中添加key |
| void remove(Key key)                          | 从集合中删除key |
| boolean contains(Key key)                     | key是否在集合中 |
| int size()                                    | 集合中键的数量   |

注意：还应实现Iterable <Key>接口，从而允许客户程序使用foreach循环访问键

通用集合的 API

与符号表一样，键的类型也不必一定要具有可比较特性。但是，处理可比较的键是常见的需求，并且允许我们支持各种基于顺序的操作，所以我们在 API 中包含了 Comparable。通过在 BST 代码中删除对 val 的引用来实现 SET 是一个简单的练习（具体参见练习 4.4.23）。另外，开发一个基于散列表的 SET 实现也不困难。

DeDup（程序 4.4.5）是一个 SET 的客户程序，从标准输入中读取一系列字符串，仅在每个字符串的第一次出现时将其输出（从而删除重复项）。在本节末尾的练习中，你可以找到更多其他 SET 客户程序的例子。

在下一节中，我们将通过案例研究说明理解这种基本抽象的重要性。

652

程序4.4.5 去重过滤器

```
public class DeDup
{
 public static void main(String[] args)
 { // 过滤掉重复的字符串
 SET<String> distinct = new SET<String>();
 while (!StdIn.isEmpty())
 { // 读取一个字符串，忽略重复
 String key = StdIn.readString();
 if (!distinct.contains(key))
 { // 保存并打印新的字符串
 distinct.add(key);
 StdOut.print(key);
 }
 StdOut.println();
 }
 }
}
```

distinct  
key     标准输入上不同  
         字符串的集合  
         当前字符串

这个SET客户程序是一个过滤器，它从标准输入读取字符串，并将字符串写入标准输出，忽略重复的字符串。为了提高效率，程序使用一个包含到目前为止遇到的不同字符串的SET。

```
% java DeDup < TaleOfTwoCities.txt
it was the best of times worst age wisdom foolishness...
```

653

**展望** 符号表实现是在算法和数据结构中进一步学习的主要内容。例如平衡二叉树、散列和 Trie 树。在 Java 和很多其他计算环境中可以找到许多这些算法和数据结构的实现。不同 API 和不同的关于键的假设需要不同的实现。算法和数据结构领域的研究者仍然在研究各种不同符号表的实现。

哪种符号表的实现更好？是散列表，还是二叉搜索树？首先考虑的要点是客户程序是否存在可比较键，是否需要符号表涉及与顺序相关的操作（如选择或排位）。如果需要，则必须使用二叉搜索树。如果不需要，大多数程序员倾向于使用散列表，因为基于散列表的符号表通常比基于二叉搜索树的符号表快（假设拥有一个对于键数据类型具有良好性能的散列函数）。

使用二叉搜索树实现符号表和集合是利用树抽象的一个绝佳的示例，树抽象无处不在。我们对日常生活中的许多树结构习以为常，包括家谱、体育比赛、公司组织结构图、语法分析树等。树也在许多计算应用程序中出现，包括函数调用树、编程语言的解析树和文件系统。树的许多重要应用植根于科学与工程，包括计算生物学中的系统发生树、计算机图形中的多维树、经济学中的极大极小博弈树以及分子动力学模拟中的四叉树。还有其他更复杂的链接结构，我们将在 4.5 节中讨论。

在日常生活中，人们可能每天都在使用字典、索引和其他类型的符号表。就在很短几年中，基于符号表的应用程序已经完全取代了电话簿、百科全书，以及各种一千年来为我们提供服务的实物。如果没有基于散列表和二叉搜索树等数据结构的符号表的实现，这些应用程序将无法实现；使用它们，我们就会有“所有的东西都可以在线立即获取”的感觉。

654

## 问答环节

问：为什么使用不可变的符号表键？

答：如果我们在散列表或 BST 中更改键，可能会使数据结构的很多属性失效。

问：为什么 HashST 中的嵌套 Node 类中的 val 实例变量被声明为 Object 类型而不是 Value 类型？

答：好问题！正如我们在 3.1 节结束时的问答中所看到的那样，Java 不允许创建泛型数组。这种限制的一个后果是需要在 get() 方法中进行强制转换，这会导致生成编译时警告（即使转换在运行时可以成功）。请注意，我们可以在 BST 中将嵌套 Node 类中的 val 实例变量声明为 Value 类型，因为它不使用数组。

问：为什么不使用 Java 库中的符号表？

答：现在你已经理解了符号表的工作原理，当然欢迎你使用工业级的 java.util.TreeMap 和 java.util.HashMap。它们遵循与 ST 相同的基本 API，但允许使用 null 键，并分别使用方法 containsKey() 和 keySet()，而不是 contains() 和 iterator()。它们还包含各种其他实用程序方法，但是它们不支持我们提到的其他一些方法，如按序遍历。你也可以使用 java.util.TreeSet 和 java.util.HashSet，其实现 API 同我们的 SET 一样。

655

## 练习

4.4.1 修改程序 Lookup 以生成一个新的程序 LookupAndPut，允许 put 操作从标准输入中获取输入的数据。遵循这样的约定：使用加号指示其后两个键入的字符串是将被插入的键 - 值对。

4.4.2 修改程序 Lookup 以生成一个新的程序 LookupMultiple，当多个值对应于同一个键的时候，将这些值存放到一个队列中（类似于程序 Index）。然后使用一个 get 请求输出所有的内容，运行过程和结果如下：

```
% java LookupMultiple amino.csv 3 0
Leucine
TTA TTG CTT CTC CTA CTG
```

- 4.4.3 修改程序 `Index` 以创建一个新的程序 `IndexByKeyword`，从命令行接收一个文件名作为参数，仅使用该文件中的关键字从标准输入中生成索引。注意：使用相同的文件作为索引，并且关键字的生成结果和程序 `Index` 一致。
- 4.4.4 修改程序 `Index` 以生成一个新的程序 `IndexLines`，仅考虑连续的字符序列作为键（不包括标点符号或数字），并使用行号代替单词位置作为值。该功能可以用于程序源代码分析：

```
% java IndexLines 6 0 < Index.java
continue 12
enqueue 15
Integer 4 5 7 8 14
parseInt 4 5
println 22
```

- 4.4.5 请为符号表 API 编写一种实现 `BinarySearchST`，维护键和值的平行数组，并按键排序。在实现 `get` 时使用二分查找法，在实现 `put` 时将大的元素向右移动一个位置以挪出空位放新数据（使用可变数组，以保持数组长度与符号表中键-值对数量的线性关系）。使用 `Index` 测试你的实现，并验证如下假说：使用这种方式实现 `Index` 时，消耗的运行时间正比于字符串的个数与输入中不同字符串的数量的乘积。
- 4.4.6 请编写符号表 API 的一种实现 `SequentialSearchST`，维护一个包含键和值的节点的链表，其顺序任意。使用程序 `Index` 测试你的实现，并验证如下假说：程序 `Index` 使用这种实现所消耗的运行时间正比于字符串的个数与输入中不同字符串的数量的乘积。
- 4.4.7 以下每个字符是一个字符串，请计算每个字符串的 `x.hashCode()%5`：

#### EASY QUESTION

按照正文中的方法，给出当序列中的第  $i$  个键与值  $i$  关联（ $i$  从 0 到 11）时的散列表。

- 4.4.8 请为程序 `HashST` 实现方法 `contains()`。
- 4.4.9 请为程序 `HashST` 实现方法 `size()`。
- 4.4.10 请为程序 `HashST` 实现方法 `keys()`。
- 4.4.11 请修改 `HashST`，增加一个带 `Key` 参数的 `remove()` 方法，用以从符号表中删除该键及对应的值（如果存在的话）。
- 4.4.12 请修改 `HashST`，使用可变数组实现，使得与每个散列值关联的链表长度在 1 到 8 之间。
- 4.4.13 当按照如下顺序将键插入一个初始为空的树后，绘制所生成的 BST：

#### EASY QUESTION

请问 BST 的高度是多少？

- 4.4.14 假设一个 BST 的键包含了 1 到 1000 之间所有的整数，请搜索 363。如下序列哪个不可能是检测过程的序列？
- |                                      |                                      |
|--------------------------------------|--------------------------------------|
| a. 2 252 401 398 330 363             | b. 399 387 219 266 382 381 278 363   |
| c. 3 923 220 911 244 898 258 362 363 | d. 4 924 278 347 621 299 392 358 363 |
| e. 5 925 202 910 245 363             |                                      |

- 4.4.15 假设一个高度为 4 的 BST 中包含下列 31 个键（按某种顺序）：

```
10 15 18 21 23 24 30 30 38 41 42 45 50 55 59
60 61 63 71 77 78 83 84 85 86 88 91 92 93 94 98
```

请绘制树最上面的三个节点（根和它的两个子节点）。

- 4.4.16 请为如下键序列绘制所有不同的 BST。

```
best of it the time was
```

656

657



- 4.4.17 判断题：给定一个 BST，假设  $x$  为叶子节点， $p$  为它的父节点。请判断：(1)  $p$  的键是 BST 中大于  $x$  的键的最小值。(2)  $p$  的键是 BST 中小于  $x$  的键的最大值。
- 4.4.18 请实现 BST 的方法 `contains()`。
- 4.4.19 请实现二分查找树的方法 `size()`。
- 4.4.20 请修改 BST，添加一个带参数 `Key` 的 `remove()` 方法，从符号表中删除该键及对应的值（如果存在的话）。提示：将键（及其关联值）替换为 BST 中小于它的最大键（及其关联值）。然后从 BST 中删除那个用来替换的节点。
- 4.4.21 使用一个递归辅助方法如 `traverse()` 为 BST 实现 `toString()` 方法。像往常一样，由于字符串拼接的代价，这种实现的时间复杂度是二次型的。加分题：请使用 `StringBuilder` 为 BST 编写一个线性时间的 `toString()` 方法。
- 4.4.22 请修改符号表 API，通过使 `get()` 方法返回一个值的迭代器，来处理一个键对应多个值的情况。按照此 API 的规定实现 BST 和 `Index`。请比较这个方法与正文中所讨论方法的优缺点。
- 658 4.4.23 请修改 BST，以实现本节最后给出的 SET API。
- 4.4.24 请修改 `HashST` 来实现本节最后给出的 SET API（从 API 中移除 `Comparable`）。
- 4.4.25 词汇索引（`concordance`）是一个按字母排序的索引表，给出了一个文本中每个单词的出现位置。因此，运行“`java index 0 0`”得到的就是一个词汇索引。曾经有一个著名的研究，一组研究人员试图通过制作一个公开的词汇索引来考证死海古卷（`Dead Sea Scrolls`）的可信度，同时保证它的细节内容不被公开。请编写一个程序 `InvertConcordance`，接收一个命令行参数  $n$ ，从标准输入中读取一个词汇索引，在标准输出中输出对应文本的前  $n$  个单词。
- 4.4.26 运行实验以验证正文中的如下假说：当使用 ST 时，`Lookup` 和 `Index` 的 `put` 操作和 `get` 请求的时间复杂度是符号表大小的对数量级。请开发一个测试客户程序，随机生成若干个键，并基于不同数据集运行测试，数据集可以来自于本书网站或自己选择。
- 4.4.27 请修改程序 BST，增加方法 `min()` 和 `max()`，用于返回表中的最小键（或最大键），如果表中不存在键，则返回 `null`。
- 4.4.28 请修改程序 BST，添加带一个参数的方法 `ceiling()` 和 `floor()`，用于返回符号表中不比给定键大（小）的最大（最小）键（如果没有这样的键存在则返回 `null`）。
- 4.4.29 请修改程序 BST，实现方法 `size()` 以返回符号表中键-值对的数量。为了实现这一功能，可以在每个节点 `Node` 内存储以该节点为根的子树的节点数量。
- 4.4.30 请修改程序 BST，增加一个方法 `rangeSearch()`，需要两个键作为参数，返回两个给定键之间所有键的迭代器。运行时间应正比于树的高度加上范围内键的数量。
- 4.4.31 请修改程序 BST，增加一个方法 `rangeCount()`，需要两个键作为参数，返回两个指定键之间所有键的数量。你的方法消耗的运行时间应该正比于树的高度。提示：首先完成上一道练习。
- 659 4.4.32 请编写一个 ST 的客户程序，创建一个符号表，用于将字母等级成绩映射到数值分数，如下表所示，然后从标准输入中读取字母等级列表，并计算其平均值（GPA）。

| A+   | A    | A-   | B+   | B    | B-   | C+   | C    | C-   | D    | F    |
|------|------|------|------|------|------|------|------|------|------|------|
| 4.33 | 4.00 | 3.67 | 3.33 | 3.00 | 2.67 | 2.33 | 2.00 | 1.67 | 1.00 | 0.00 |

660

## 二叉树练习

以下练习的目的是增强读者对于二叉树（不一定是 BST）的理解。习题中都假设一个 `Node` 类具有三个实例变量：一个 `double` 型变量（总是正数）和两个 `Node` 的引用，与链表一样，你会发现最有用的方法是使用正文中描述的可视化表示方法绘制图形。

4.4.33 实现如下函数，每个函数的参数都是一个 Node 类型，用于表示二叉树的根。

```
int size() 树中的节点数
int leaves() 链接都为 null 的节点的数量
double total() 所有节点的键值之和
```

你的方法都应该在线性时间内运行。

4.4.34 请实现一个线性运行时间的函数 height(), 对于所有根节点到叶节点 (单节点树的高度为 0) 的路径, 返回节点数量最多的那一条路径的节点数量。

4.4.35 如果一个二叉树根节点的键大于所有后代的键, 则称二叉树是堆有序的。请实现一个线性运行时间的方法 heapOrdered(), 如果树是堆有序的, 则返回 true, 否则返回 false。

4.4.36 一棵二叉树为平衡树的条件是其两棵子树是平衡树且其两棵子树的高度最多相差 1。请实现一个线性运行时间的方法 balanced(), 如果树是平衡树, 则返回 true; 否则返回 false。

4.4.37 当两棵二叉树具有相同的形状, 仅仅是键值不同时, 我们称其为同构 (isomorphic)。请实现线性运行时间静态方法 isomorphic(), 以两棵树的引用作为参数, 如果两棵树为同构则返回 true, 否则返回 false。然后, 实现一个线性运行时间的静态方法 eq(), 以两棵树的引用作为参数, 如果它们引用的是一样的树 (同构且值相同), 则返回 true, 否则返回 false。

4.4.38 请实现一个线性运行时间的函数 isBST(), 如果一棵二叉树是 BST 则返回 true, 否则返回 false。

661

答案: 这个任务比表面看起来更复杂。使用一个递归辅助函数 isBST(), 增加两个参数 lo 和 hi, 如果二叉树为 BST 并且所有值在 lo 和 hi 之间, 则返回 true。使用 null 来表示可能的最大和最小键值。

```
public static boolean isBST()
{ return isBST(root, null, null); }

private boolean isBST(Node x, Key lo, Key hi)
{
 if (x == null) return true;
 if (lo != null && x.key.compareTo(lo) <= 0) return false;
 if (hi != null && x.key.compareTo(hi) >= 0) return false;
 if (!isBST(x.left, lo, x.key)) return false;
 if (!isBST(x.right, x.key, hi)) return false;
}
```

4.4.39 请编写一个函数 levelOrder(), 以层序输出 BST 的键: 首先输出根; 然后从左到右输出根下一层的所有节点; 然后是根节点下两层的所有节点 (从左到右), 以此类推。提示: 使用 Queue<Node>。

4.4.40 阅读下面的程序, 计算针对某些二叉树的返回结果。然后估计此段程序对于任意二叉树的执行结果, 并证明。

```
public int mystery(Node x)
{
 if (x == null) return 0;
 return mystery(x.left) + mystery(x.right);
}
```

答案: 任何二叉树都返回 0。

662

## 创新练习

4.4.41 拼写检查。请编写一个 SET 的客户程序 SpellChecker, 从命令行接收一个包含单词字典的文件名作为参数, 然后从标准输入读取字符串, 输出那些在字典中不存在的字符串。你可以在本书网站找到一个字典文件。加分题: 扩展你的程序以处理常见的后缀, 如 -ing 或 -ed。

- 4.4.42 拼写更正。编写一个 ST 的客户程序 `SpellCorrector`，该程序是一个过滤器，用于提示并建议替换标准输入中常见拼写错误的单词，并把结果写入标准输出。从命令行接收包含常见拼写错误和更正的文件作为参数。在本书网站可以找到一个示例。
- 4.4.43 Web 过滤器。编写一个 SET 客户程序 `WebBlocker`，从命令行接收一个包含不良网站列表的文件作为参数，然后从标准输入中读取字符串，并仅输出不在列表中的网站。
- 4.4.44 集合操作。请在 SET 中增加两个方法 `union()` 和 `intersection()`，以两个集合作为参数，并分别返回这两个集合的并集和交集。
- 4.4.45 频率符号表。请开发一个数据类型 `FrequencyTable`，支持以下操作：`click()` 和 `count()`。两个函数均带有字符串参数。数据类型中需要记录使用给定字符串作为参数调用 `click()` 的次数，每次调用 `click()` 操作会使相应的计数递增 1，`count()` 操作则返回计数器的值（可能为 0）。该数据类型可以用于 Web 流量分析器、统计每首歌播放次数的音乐播放器、统计呼叫次数的电话软件等。
- 4.4.46 一维范围搜索。请开发一个数据类型以支持如下操作：插入一个日期、查找一个日期、统计数据结构中位于特定区间的日期数量。使用 Java 的 `java.util.Date` 数据类型。
- 663 4.4.47 非重叠区间搜索。给定一个非重叠整数区间列表，请编写一个函数，该函数需要一个整数参数，确定该整数所处的区间（如果存在）。例如，如果区间是 1643~2033、5532~7643、8999~10332 和 5666653~5669321，则查询点 9122 位于第三个区间中，而 8122 不在任何区间中。
- 4.4.48 查询 IP 所属国家。编写一个 BST 的客户程序，使用本书网站上的数据文件 `ip-to-country.csv`，确定一个给定的 IP 地址来自哪个国家。该数据文件包含 5 个字段：开始 IP 地址范围，结束 IP 地址范围，两个字符的国家代码，三个字符的国家代码和国家名称。IP 地址不重叠。这种数据库工具可用于信用卡欺诈检测、垃圾邮件过滤、网站自动语言选择以及网站服务器日志分析。
- 4.4.49 网页的反向索引。给定一个网页列表，创建一个包含网页中所有单词的符号表。把每一个单词与出现该单词的网页列表关联起来。编写一个程序，读取一个网页列表，创建一个符号表，支持单个单词查询，并返回出现过查询单词的网页列表。
- 4.4.50 网页的反向索引。扩展前面的练习，以便其支持多个单词查询。在这样的情况下，输出包含每个查询单词都出现了不少于一次的网页列表。
- 4.4.51 多词搜索。编写一个程序，从命令行读取  $k$  个单词，从标准输入中读入一串文本，从这串文本中找出包含所有这  $k$  个单词的最小文本区间（不一定是相同的顺序）。提示：将输入的文本串按单词分割，每一个赋予一个下标，对于每个下标  $i$ ，找到包含  $k$  个查询词的最小区间  $[i, j]$ 。对于  $k$  个查询词，记录在这个区间中每个词出现的次数。给定  $[i, j]$ ，通过递减单词  $i$  的计数器来计算  $[i+1, j]$ 。然后，递增  $j$  直到在这个区间中这  $k$  个单词都出现了至少一次（或者说直到 `Word[i]` 再次出现）。
- 4.4.52 在国际象棋中重复和局。在国际象棋游戏中，如果要移动子一方连续出现了三次一样的棋子布局，则移动的这一方就可以宣布和棋。描述如何使用计算机程序识别这种情况。
- 664 4.4.53 教务安排。在著名的东北大学（美国），教务处曾经安排一位教师在同一时间内给两个不同的班级授课。请设计一种方法来检查这种冲突，帮助教务处今后避免犯这种错误。为了简单起见，假定所有的课程均为 50 分钟，且开始时间为 9 点、10 点、11 点，以及下午 1 点、2 点、3 点。
- 4.4.54 随机元素。请在程序 BST 中添加一个方法 `random()`，返回一个随机键。提示：可以在每个节点中保存子树的大小（请参阅练习 4.4.29）。运行时间应与树的高度成正比。

- 4.4.55 层序统计。在 BST 中添加一个方法 `select()`，接收一个整数参数 `k`，并返回 BST 中第 `k` 个最小键。在每个节点中保存子树的大小（具体参见本节练习 4.4.29）。程序运行时间应与树的高度成正比。
- 4.4.56 排名查询。请在程序 BST 中添加一个 `rank()` 方法，接收一个键作为参数，返回 BST 中严格小于该键的键的数量。在每个节点中保存子树的大小（具体参见练习 4.4.29）。程序运行时间应与树的高度成正比。
- 4.4.57 广义队列。请实现一个支持以下 API 的类，它扩展了队列和栈，支持删除最早插入的第 `i` 个项目（请参阅练习 4.3.40）：

| public class GeneralizedQueue<Item> |                                         |
|-------------------------------------|-----------------------------------------|
| <code>GeneralizedQueue()</code>     | 创建一个空的广义队列                              |
| <code>boolean isEmpty()</code>      | 广义队列是否为空                                |
| <code>void add(Item item)</code>    | 将项目 <code>item</code> 插入广义队列中           |
| <code>Item remove(int i)</code>     | 从广义队列中移除并返回第 <code>i</code> 个最早插入队列中的项目 |
| <code>int size()</code>             | 队列中项目的数量                                |

通用广义队列的 API

665

使用一个 BST，把第 `k` 个插入的元素与键 `k` 关联起来，并在每个节点保存以该节点为根节点的子树中节点的总数。为了查找最近插入的第 `i` 个项目，需要在 BST 中搜索第 `i` 个最小的项。

- 4.4.58 稀疏向量。如果一个  $d$  维向量中非零值的数量很少，则称为稀疏向量。你的目标是设计一种向量表示方法，所使用的空间正比与其非零值个数，且可以实现两个稀疏向量的加法，要求其运行时间正比于非零值个数之和。实现一个支持以下 API 的类：

| public class SparseVector                      |                                                              |
|------------------------------------------------|--------------------------------------------------------------|
| <code>SparseVector()</code>                    | 创建一个稀疏向量                                                     |
| <code>void put(int i, double v)</code>         | 将稀疏向量 <code>a</code> 的第 <code>i</code> 个元素设置为 <code>v</code> |
| <code>double get(int i)</code>                 | 返回稀疏向量 <code>a</code> 的第 <code>i</code> 个元素                  |
| <code>double dot(SparseVector b)</code>        | 稀疏向量的点积                                                      |
| <code>SparseVector plus(SparseVector b)</code> | 稀疏向量的向量和                                                     |

由 double 值构成的稀疏向量的 API

- 4.4.59 稀疏矩阵。如果一个  $n \times n$  矩阵的非零值个数正比于  $n$ （或更少），则称为稀疏矩阵。你的目标是设计一种矩阵表示方法，所使用的空间与其非零值个数成正比，并且能够实现两个稀疏矩阵相加，要求其运行时间正比于非零值个数之和（可能还有一个额外的  $\lg n$  因子）。实现一个支持以下 API 的类：

| public class SparseMatrix                       |                                                                                |
|-------------------------------------------------|--------------------------------------------------------------------------------|
| <code>SparseMatrix()</code>                     | 创建一个稀疏矩阵                                                                       |
| <code>void put(int i, int j, double v)</code>   | 将稀疏矩阵 <code>a</code> 的第 <code>i</code> 行第 <code>j</code> 列元素设置为 <code>v</code> |
| <code>double get(int i, int j)</code>           | 返回稀疏矩阵 <code>a</code> 的第 <code>i</code> 行第 <code>j</code> 列元素                  |
| <code>SparseMatrix plus(SparseMatrix b)</code>  | 稀疏矩阵的加法                                                                        |
| <code>SparseMatrix times(SparseMatrix b)</code> | 稀疏矩阵的乘法                                                                        |

由 double 值构成的稀疏矩阵的 API

666

- 4.4.60 无重复队列。请创建一个队列数据类型，任意时刻在队列中一个项目至多可以出现一次。如果某个元素已经在队列中存在，则忽略其插入请求。

- 4.4.61 可变字符串。请创建一个数据类型，支持字符串如下表所示的 API。请使用一个 BST 并采用对数型时间实现所有操作。

| public class MutableString |                     |
|----------------------------|---------------------|
| MutableString()            | 新建一个空的可变字符串         |
| char get(int i)            | 返回可变字符串对象中的第i个字符    |
| void insert(int i, char c) | 将字符c插入可变字符串对象的第i个位置 |
| void delete(int i)         | 删除可变字符串对象的第i个字符     |
| int length()               | 返回可变字符串的长度          |

#### 可变字符串的 API

- 4.4.62 赋值语句。请编写一个程序，解析包含赋值语句的程序并对其求值，并将语句转化成全括号型算术表达式语句（参见程序 4.3.5）。例如，给定输入：

```
A = 5
B = 10
C = A + B
D = C * C
print(D)
```

你的程序应该打印出值 225。假设所有的变量和值都是 double 类型的。请使用一个符号表保存并跟踪变量名。

- 4.4.63 熵。我们定义一个包含  $n$  个单词（其中  $k$  个互不相同）的文本语料库的相对熵如下：

$$E = 1/(n \lg n) (p_0 \lg(k/p_0) + p_1 \lg(k/p_1) + \cdots + p_{k-1} \lg(k/p_{k-1}))$$

其中  $p_i$  是单词  $i$  出现次数的百分率。请编写一个程序，读取一个文本语料库，输出其相对熵。请将所有的字母转换为小写字母，并把标点符号都看作空格。

- 4.4.64 动态离散分布。创建一个数据类型，支持以下两个操作：add() 和 random()。add() 方法插入一个新项到数据结构中，如果该项不存在则插入；否则，把其频率计数递增 1。random() 方法随机返回一个项，其概率按照每个元素的频率加权。在每个节点中保存子树的大小（请参阅练习 4.4.29）。程序运行时间应与树的高度成正比。
- 4.4.65 股票账户。在 StockAccount（程序 3.2.8）中实现 buy() 和 sell() 两种方法。使用符号表来存储每只股票的股数。
- 4.4.66 密码子使用表。请编写一个程序，使用符号表输出来自标准输入的基因组密码子的概率统计信息（每千个的频率），如下所示：

```
UUU 13.2 UCU 19.6 UAU 16.5 UGU 12.4
UUC 23.5 UCC 10.6 UAC 14.7 UGC 8.0
UUA 5.8 UCA 16.1 UAA 0.7 UGA 0.3
UUG 17.6 UCG 11.8 UAG 0.2 UGG 9.5
CUU 21.2 CCU 10.4 CAU 13.3 CGU 10.5
CUC 13.5 CCC 4.9 CAC 8.2 CGC 4.2
CUA 6.5 CCA 41.0 CAA 24.9 CGA 10.7
CUG 10.7 CCG 10.1 CAG 11.4 CGG 3.7
AUU 27.1 ACU 25.6 AAU 27.2 AGU 11.9
AUC 23.3 ACC 13.3 AAC 21.0 AGC 6.8
AUA 5.9 ACA 17.1 AAA 32.7 AGA 14.2
AUG 22.3 ACG 9.2 AAG 23.9 AGG 2.8
GUU 25.7 GCU 24.2 GAU 49.4 GGU 11.8
GUC 15.3 GCC 12.6 GAC 22.1 GGC 7.0
GUA 8.7 GCA 16.8 GAA 39.8 GGA 47.2
```

- 4.4.67 长度为  $k$  的唯一字符串子串。编写一个程序，从命令行读取参数  $k$ ，从标准输入读入文本，并计算该文本中给定长度  $k$  的唯一子字符串的数量。例如，如果输入是 GCCGGGCGCG，则有 5 个长度为 3 的唯一子字符串：CGC、CGG、GCG、GGC 和 GGG。这种计算适用于数据压缩。提示：使用字符串方法 `substring( $i$ ,  $i+k$ )` 来抽取第  $i$  个子字符串并插入符号表中。本书官网提供了两个数据集，即一个大基因组以及  $\pi$  的前一千万位数字，用它们来测试你的程序。
- 4.4.68 随机电话号码。编写一个程序，从命令行接收整数参数  $n$ ，输出  $n$  个随机电话号码，格式为  $(\text{XXX})\text{XXX-XXXX}$ 。使用集合来避免多次重复选择相同的号码。请使用合法的区号代码（本书官网上提供合法区号代码的文件）。
- 4.4.69 密码检查器。编写一个程序，从命令行接收一个字符串参数，并从标准输入中读取一个单词字典，然后检查字符串是否为一个“好”的密码。在这里，假定“好”的密码意味着：（1）至少八个字符的长度；（2）不是字典中的一个单词；（3）不是字典中的一个单词加数字 0~9（如 `hello5`）；（4）不是用数字分隔的两个单词（如 `hello2world`）；（5）再次检查第 2~4 个条件，确认没有出现字典中单词的倒序。

669

## 4.5 案例研究：小世界现象

我们用数学模型图（graph）来研究实体间成对性质。图能够帮助我们研究自然世界，也对我们更好地理解和改进我们创建的网络非常重要。在过去的一个世纪里，从神经生物学的神经系统模型，到医学传染病的传播，再到电话系统的发展，图在科学和工程领域发挥了重要的作用，其中也包括了对互联网发展的推动。

某些图显示了一个被称为小世界现象（small-world phenomenon）的特定属性。你可能熟悉这个属性，有时它也被称为六度分隔（six degrees of separation）。其基本思想是：即使我们每个人认识的熟人相对较少，但想在世界上任意两个陌生人之间建立联系，所需要的人际链条并不长（六度分隔）。这一假设在 20 世纪 60 年代由斯坦利·米尔格拉姆（Stanley Milgram）通过实验验证，并在 20 世纪 90 年代由邓肯·瓦茨（Duncan Watts）和斯蒂芬·斯特罗格茨（Stephen Strogatz）建立了数学模型。近年来，这一原理在各种各样的应用中被证明是非常重要的。科学家对小世界图感兴趣，因为它们模拟了自然现象；工程师对此也有兴趣，因为利用小世界图的自然属性可以更好地构建网络。

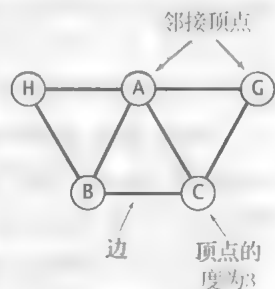
在本节中，我们将讨论小世界现象和图的基本计算问题。事实上，这个问题：

一个既定的图是否表现出小世界现象？

带来的重大计算任务是巨大的。为了解决这个问题，我们将研究一个图处理所需的数据类型和几个有用的图处理客户程序。具体来说，我们将考察一个计算最短路径（shortest paths）的客户程序，这个计算本身有着很多重要的应用。

本节另一个主题是我们一直在研究的算法和数据结构，它们在图处理中起着核心作用。事实上，你会发现本章前面介绍的几种基本数据类型可以帮助我们开发优雅、高效的代码，以研究图的属性。

**图** 为了避免术语混淆，我们现在给出一些定义。图包含顶点（vertex）集合和边（edge）的集合。每条边表示两个顶点之间的连接。如果两个顶点通过边相连，则两个顶点相邻（adjacent），顶点的度数（degree）是该顶点的邻接顶点（或邻居）的数量。请注意，这里所说的图与函数图像（函数值的图像）或制图（绘画）的概念之间没有关系。我们经常通过绘制由线（边）连接带有标记的圆（顶点）来可视



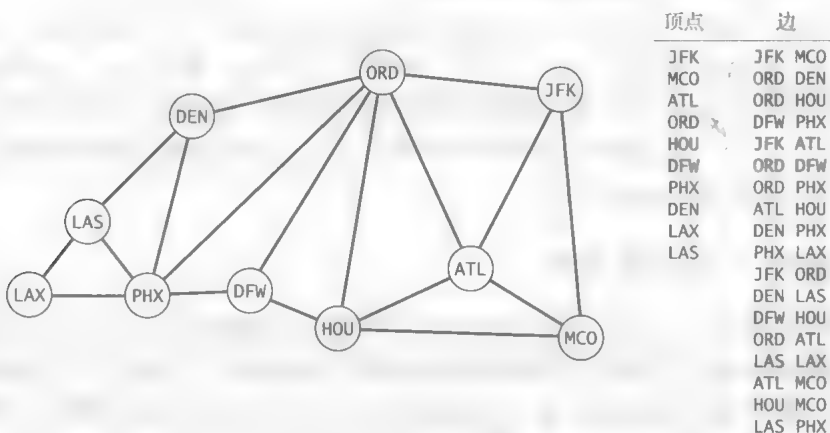
图术语

670

化地表示一幅图，但是要记住连接本身才是重要的，而非我们描绘它们的方式。

下面我们罗列了不同领域的各类系统，它们当中都有图的应用，这是我们理解图结构的最佳出发点。

交通系统（transportation systems）。火车轨道连接站，道路连接交叉口，航空公司航线连接机场……所有这些系统都可以构建成简单的图模型。毫无疑问，你肯定使用过基于此类模型的应用程序。例如，从互动式地图程序或GPS设备获取路线，或使用在线服务进行旅行预订，你常常关心的问题是：从这里到那里的最佳途径是什么？



交通系统的图模型

人类生物学（human biology）。动脉和静脉连接器官，突触连接神经元，关节连接骨骼，因此对人体生物学的理解取决于如何建立适当的图模型。也许这个领域最大也是最重要的建模挑战是人脑建模。也就是说，神经元之间的局部连接如何转化为意识、记忆和智力？

社交网络（social network）。人与人之间存在着各种各样的联系，从对传染病的研究到对政治倾向的研究，这些关系的图模型对于我们理解它们的影响至关重要。同时，信息如何在在线社交网络中传播的问题也吸引了很多研究者。

物理系统（physical system）。原子相互连接形成分子，分子相互连接形成物质或晶体，粒子通过重力或磁力等相互作用力连接。例如，图模型也适用于研究2.4节中考虑的渗透问题。当系统演化发展时，局部作用如何传播？

通信系统（communications system）。从电路到电话系统，到互联网，再到无线服务，通信系统都是基于设备的连接而建立的网络。至少在过去的一个世纪里，图模型在这样的系统开发中起到了关键的作用。工程师经常需要知道什么是设备连接的

| 系统   | 顶点  | 边    |
|------|-----|------|
| 自然现象 |     |      |
| 血液循环 | 器官  | 血管   |
| 骨骼   | 关节  | 骨头   |
| 神经   | 神经元 | 突触   |
| 社会   | 人   | 人际关系 |
| 流行病学 | 人   | 感染   |
| 化学   | 分子  | 化学键  |
| 多体模拟 | 粒子  | 作用力  |
| 遗传学  | 基因  | 变异   |
| 生物化学 | 蛋白质 | 相互作用 |
| 工程系统 |     |      |
| 交通   | 机场  | 路线   |
|      | 路口  | 道路   |
| 通信   | 电话  | 电话线  |
|      | 计算机 | 电缆   |
|      | 网页  | 链接   |
| 资源分配 | 电站  | 电线   |
|      | 用户  |      |
|      | 水库  | 水管   |
|      | 库房  |      |
|      | 用户  | 卡车路线 |
|      | 零售店 |      |
| 机械   | 结合点 | 横梁   |
| 软件   | 模块  | 调用   |
| 财务   | 账户  | 交易   |

典型的图模型



最佳方式。

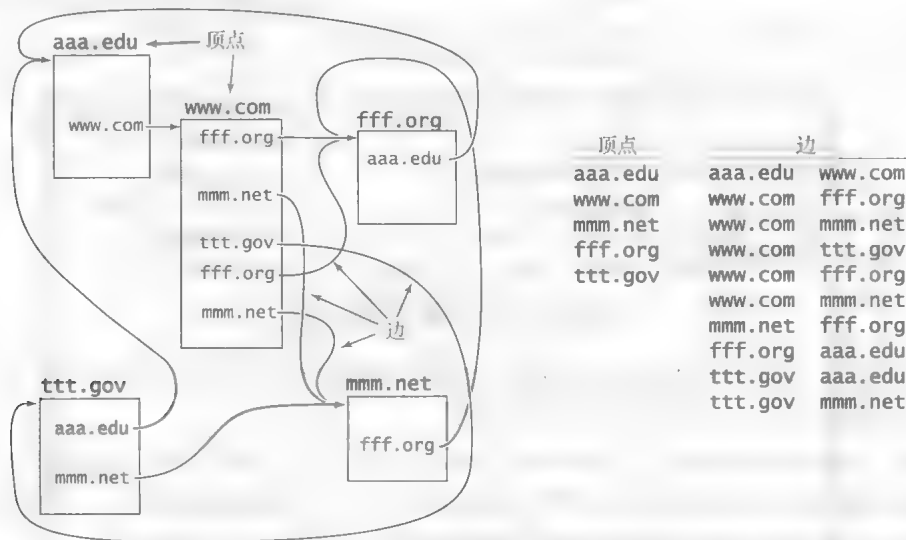
672

资源分配 (resource distribution)。电力线路连接发电站和家庭电气系统，管道连接水库和家庭供水，卡车路线连接仓库和零售店。研究资源分配的有效和可靠的手段取决于准确的图模型。例如，分配系统的瓶颈在哪里？

机械系统 (mechanical system)。桁架或钢梁连接桥梁或建筑物的接缝。图模型帮助我们设计这些系统并理解其特性。例如，哪个力必须由一个结合点或一个横梁承受？

软件系统 (software system)。一个程序模块中的方法调用其他模块中的方法。正如我们在本书中所看到的那样，理解这种关系是软件设计成功的关键。如果在 API 中做出修改，哪些模块会受到影响？

金融体系 (financial system)。交易连接账户，账户将客户连接到金融机构。这些仅仅是人们用来研究复杂金融交易的图表模型中的一小部分，并有助于更好地理解金融交易。例如，哪些交易是例行的，哪些标志着可能转化为利润的重要事件？



互联网的图模型

673

这些模型中的一部分是自然现象的模型，我们的目标是通过开发简单的模型来获得对自然世界的更好理解，然后用它们来形成假设以便我们进行测试。其他图模型是我们自己设计的网络，我们的目标是设计一个更好的网络，或者通过了解网络的基本特性来更好地维护网络。

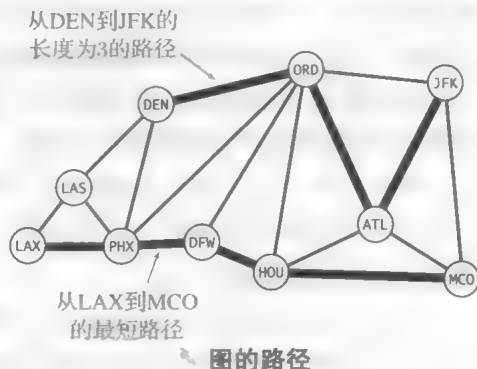
无论大小，图都是很实用的模型。一个图即便只有几十个顶点和边（例如一个化合物的模型图，其中顶点是分子，边是化学键），也已经算是一个复杂的组合对象，因为这些顶点和边已经可以组合出成千上万种图，因此理解某些特定的图的结构和特点是很重要的。具有数以亿计或万亿计个顶点和边的图（例如包含所有电话呼叫数据的政府数据库，或人类神经系统的图模型）非常复杂，对它们的研究是一项重大的计算挑战。

图的处理通常涉及通过文件中的信息构建图和回答关于图的问题。除了刚刚提到的示例中的特定应用的问题外，我们经常需要询问有关图的基本问题。图有多少个顶点和边？一个给定顶点的邻接顶点有哪些？另外，还有些问题取决于对图的结构的理解。例如，图中的路径 (path) 是由边连接的相邻顶点的序列。是否存在连接两个给定顶点的路径？连接两个顶点的最短路径的长度（边的数量）是多少？在本书中我们已经看到了几个比这些更复杂的科学应用问

题的例子。随机游走落在每个点的概率是多少？某个图所代表的系统渗透的概率是多少？

当你在后面的课程中遇到复杂的系统时，你肯定会在许多不同的应用场景中遇到图。你也可以在数学、运筹学或计算机科学的后续课程中详细研究其属性。图处理中的一部分问题相对简单，可以用我们一直在学习的这些数据类型的实现方法解决；还有一部分问题提出了难以克服的计算挑战，科学家们还在寻求高效的解决方案。

**图数据类型** 图处理算法通常首先通过添加边来构建一个图的内部表示，然后通过遍历顶点和与给定顶点的邻接顶点来处理它。以下是支持这些处理操作所需的 API：



```
public class Graph
{
 Graph()
 Graph(In in, String delimiter)
 void addEdge(String v, String w)
 int V()
 int E()
 Iterable<String> vertices()
 Iterable<String> adjacentTo(String v)
 int degree(String v)
 boolean hasVertex(String v)
 boolean hasEdge(String v, String w)
}
```

创建一个空图  
从输入流中读取图  
增加v-w边  
顶点数  
边数  
图中的顶点  
v的邻接顶点  
v的邻接顶点的数量  
v是图中的顶点吗？  
v-w是图中的边吗？

#### 顶点为 String 类型的图的 API

像往常一样，这个 API 反映了几种设计选择，每种设计选择其实都可以有多种不同的方案，我们现在对其中的一部分进行简要讨论。

**无向图 (undirected graph)**。边是无向 (undirected) 的，也就是从 v 连接到 w 的边和从 w 连接到 v 的边是一样的。我们的兴趣在于连接本身，而不是连接的方向。有向边（如地图中的单行道）需要稍微不同的数据类型（请参见练习 4.5.41）。

**字符串顶点类型**。我们当然可以用一个泛型来做顶点类型，从而使得客户程序可以用任何类型的对象来构建图。然而，这种结果的代码会变得非常冗长，我们把相关实现留作练习（参见练习 4.5.9）。对于我们在这里的应用，字符串顶点类型就足够了。

**无效的顶点名称 (invalid vertex name)**。如果方法 adjacentTo()、degree() 和 hasEdge() 的字符串参数不能与任何一个顶点名称相匹配，则会抛出异常。客户程序可以调用 hasVertex() 来检测这种情况。

**隐式顶点创建 (implicit vertex creation)**。当一个字符串被用作 addEdge() 的参数时，我们假定它是一个顶点名称。如果尚未添加使用该名称的顶点，则我们的实现会添加这样一个顶点。addVertex() 方法的一种替代设计方案是由客户程序主动创建顶点，那么就会需要更多的客户程序代码（来创建顶点）和更烦琐的实现代码（来检查连接先前创建的顶点的边）。

**自环和平行边 (self-loops and parallel edge)**。尽管 API 没有明确地解决这个问题，但是我们假定实现确实允许自环（将一个顶点的边连接到顶点自身），但不允许平行边（相同边的两个副本）。检查是否存在自环和平行边是很容易的；因此我们选择忽略这两个检查。

**客户程序查询方法 (client query method)**。我们的 API 中还包含方法 V() 和 E()，以向客户程序提供图中顶点和边的数量。类似地，方法 degree()、hasVertex() 和 hasEdge() 在客

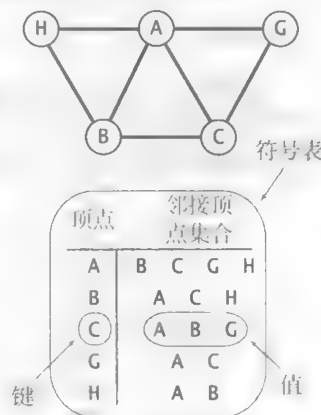
户程序代码中也都很实用。我们把这些方法的实现作为练习放在后面，但目前我们假定它们已经存在于我们的 Graph API 中。

这些设计决策都不是“神圣不可侵犯的”；它们只是我们为本书中的代码所做的选择。有一些选择可能需根据应用场景而改变，而另一些决策可能要在具体实现时做出选择。因此，选择这个设计决策的原因至关重要，值得仔细思考，并在做出选择后做好准备捍卫它们。

Graph（程序 4.5.1）实现这个 API。它的内部表示是符号表型集合（symbol table of sets），键是顶点，值是邻接顶点的集合。这种表示使用了我们在 4.4 节介绍的两种数据类型 ST 和 SET。它有以下三个重要的属性：

- 客户程序可以高效地遍历图的顶点。
- 客户程序可以高效地遍历顶点的邻接顶点。
- 内存使用量与边的数量成正比。

这些属性遵循了 ST 和 SET 的基本属性。如你所见，这两个迭代器是图处理的核心。



符号表集合的图表示

程序4.5.1 图数据类型

```
public class Graph
{
 private ST<String, SET<String>> st;

 public Graph()
 { st = new ST<String, SET<String>>(); }

 public void addEdge(String v, String w)
 { // 把v放到w的SET中，把w放到v的SET中
 if (!st.contains(v)) st.put(v, new SET<String>());
 if (!st.contains(w)) st.put(w, new SET<String>());
 st.get(v).add(w);
 st.get(w).add(v);
 }

 public Iterable<String> adjacentTo(String v)
 { return st.get(v); }

 public Iterable<String> vertices()
 { return st.keys(); }

 // V()、E()、degree()、hasVertex()、hasEdge()的实现参见练习4.5.1~4.5.4

 public static void main(String[] args)
 { // 从标准输入读取边；打印出结果图
 Graph G = new Graph();
 while (!StdIn.isEmpty())
 G.addEdge(StdIn.readString(), StdIn.readString());
 StdOut.print(G);
 }
}
```

这个实现使用ST和SET（见4.4节）来实现图数据类型。客户程序通过添加边来构建图，并通过迭代访问顶点以及与每个顶点相邻的顶点集对其进行处理。它能够从输入流读入一个图。请参阅正文中关于toString()和与之匹配的构造器（从输入流读入图）的解释。

```
% more tinyGraph.txt
A B
A C
C G
A G
H A
B C
B H
```

```
% java Graph < tinyGraph.txt
A B C G H
B A C H
C A B G
G A C
H A B
```

作为客户程序代码的简单示例，我们分析如何打印 Graph 的问题。一种自然的方式是打印顶点列表，以及每个顶点的邻接顶点列表。我们使用如下方法来实现 Graph 中的 toString():

```
public String toString()
{
 String s = "";
 for (String v : vertices())
 {
 s += v + " ";
 for (String w : adjacentTo(v))
 s += w + " ";
 s += "\n";
 }
 return s;
}
```

这个代码将每条边打印了两次——发现 w 是 v 的邻接顶点打印一次，发现 v 是 w 的邻接顶点打印一次。许多图算法都把这种处理图中每条边的方式作为基本范例，但务必要记得“每条边均处理了两次”！像往常一样，这个实现仅适用于小型图，因为字符串连接是线性时间，因此这段代码的运行时间与字符串长度的平方成正比。

刚才所分析的输出格式定义了一个合理的文件格式：每一行都是一个顶点名称，后面跟着该顶点的邻接顶点名称。因此，我们的图的基本 API 中包含了一个构造函数，用于以这种格式（带有邻接顶点的顶点列表）从输入流中构建图。为了灵活，我们支持在顶点名之间使用专用的分隔符（不必一定采用空格当分隔符，因此顶点名称可能包含空格），如下所示：

```
public Graph(In in, String delimiter)
{
 st = new ST<String, SET<String>>();
 while (in.hasNextLine())
 {
 String line = in.readLine();
 String[] names = line.split(delimiter);
 for (int i = 1; i < names.length; i++)
 addEdge(names[0], names[i]);
 }
}
```

678

现在我们将看到，将此构造函数和 toString() 添加到 Graph，就构成了一个完整的数据类型，可以适用于各种应用场景。请注意，当输入是一个边的列表，每行代表一条边时，这个相同的构造函数（使用空格为分隔符）也可以正常工作，就像程序 4.5.1 的测试客户程序一样。

**图客户程序的例子** 作为第一个图处理的客户程序，我们考虑一个社交关系的例子，这个例子对你来说非常熟悉，而且可以随时获取大量的扩展数据。

在本书网站上，你可以找到文件 movies.txt（以及许多类似的文件），其中包含电影列表和出现在其中的演员。每一行都会给出一个电影的名字，然后是演员名单（出现在该电影中的表演者的名字列表）。由于名字中包含空格和逗号，因此“/”字符被用作分隔符（现在你应该明白为什么我们的第二个 Graph 构造函数将分隔符也作为参数）。

如果你研究 movies.txt，你将注意到一些特性，尽管它们很小，但在使用这些数据时需要注意：



同一部电影中，是不是应该用顶点表示演员，再用边把他们连接起来？两种选择都是合理的，但是我们应该使用哪一种呢？该决定将影响客户程序和实现代码。另一种处理方法（我们选择它是因为它的实现代码更简单）是电影和演员都作为顶点，通过边将每个电影连接到该电影中的每个演员。正如你所看到的，处理这个图的程序可以回答各种有趣的问题。IndexGraph（程序4.5.2）是第一个例子，它接受一个查询，比如一个电影的名字，并打印出现在该电影中的演员的列表。

程序4.5.2 使用图来反转索引

```
public class IndexGraph
{
 public static void main(String[] args)
 { // 构造一个图，然后执行查询
 In in = new In(args[0]);
 String delimiter = args[1];
 Graph G = new Graph(in, delimiter);
 while (StdIn.hasNextLine())
 { // 读取一个顶点，然后打印它的邻接顶点
 String v = StdIn.readLine();
 for (String w : G.adjacentTo(v))
 StdOut.println(" " + w);
 }
 }
}
```

|           |        |
|-----------|--------|
| in        | 输入流    |
| delimiter | 输入分隔符  |
| G         | 图      |
| v         | 查询     |
| w         | v的邻接顶点 |

这个Graph客户程序从命令行指定的文件创建一个图，然后从标准输入读取顶点名称并打印它的邻接顶点。当文件对应一个电影-演员表时，我们就会得到一个二部图，这个程序相当于一个交互式的反转索引。

```
% java IndexGraph tinyGraph.txt " "
C
A
B
G
A
B
C
G
H
```

```
% java IndexGraph movies.txt "/"
Da Vinci Code, The (2006)
Aubert, Yves
...
Herbert, Paul
...
Wilson, Serretta
Zaza, Shane
Bacon, Kevin
Animal House (1978)
Apollo 13 (1995)
...
Wild Things (1998)
River Wild, The (1994)
Woodsman, The (2004)
```

输入一个电影名称并获得它的演员表只不过是在 movies.txt 中返回对应的行（尽管 IndexGraph 打印的演员列表是按照姓氏排列的，因为这是 SET 提供的默认迭代顺序）。IndexGraph 更有趣的功能是你可以键入一个演员的名字，并获得该演员出演的电影列表。为什么能实现这个功能呢？即使 movies.txt 看似只是将电影连接到演员，但是图中的边也是将演员连接到电影的连接。

如果所有顶点可以分成两个不相交的集合，所有连接都将一个集合的顶点连接到另一集合，这样图被称为二部图（bipartite graph）（又称二分图、偶图等——译者注）。正如这个例子所说明的，二部图有许多自然的特性，我们经常可以以有趣的方式加以利用。

正如我们在 4.4 节的开头所看到的那样，索引范式是通用而且非常常见的。值得反思的

是，构建一个二部图提供了一个简单的方法来自动反转任何索引！文件 `movies.txt` 由电影来索引，但我们也可以通过演员查询。你可以以完全相同的方式使用 `IndexGraph` 来处理 4.2 节开头讨论的任何索引数据，如打印出现在给定页面上的索引词或与给定氨基酸相对应的密码子。由于 `IndexGraph` 将分隔符作为命令行参数，因此可以使用它为 `.csv` 创建交互式的反转索引。

这种反转索引功能是图数据结构（`graph data structure`）的直接优势。接下来，我们从处理数据结构的算法中研究一些间接优势。

**图的最短路径** 给定图中的两个顶点，路径（`path`）是连接它们的一系列边。最短路径（`shortest path`）是在所有这些路径上长度（`length`）或距离（`distance`，边的数量）最小的路径（通常有多条最短路径）。寻找连接图中两个顶点的最短路径是计算机科学中的一个基本问题。最短路径已经成功地广泛应用于解决大规模问题，从互联网路由到金融交易，再到大脑神经元的动态。

举个例子，假设你是一个假想的廉价航空公司的客户，这个航空公司服务于数量有限的城市，并且航线数量也有限。假设从一个地方到另一个地方的最佳途径是尽可能减少你的航段数量，因为航班延误可能会使旅途更长。最短路径算法正是你计划旅行所需要的。这样理解基本问题并且探索解决这个问题的方法，符合我们的直觉。但接下来在这个例子的背景中，我们将考虑一个更加抽象的图模型。

根据应用，客户对于最短路径有各种各样的需求。我们想要得到的是连接两个给定顶点的最短路径，还是只是这样一条路径的长度？我们是否会有大量的这种查询？有没有特别感兴趣的特定顶点？在大型图中或者面对大量的查询时，我们必须特别注意这些问题，因为计算最短路径的代价被证明是非常巨大的。我们从下面的 API 开始：

|                                                      |                     |  |
|------------------------------------------------------|---------------------|--|
| <code>public class Pathfinder</code>                 |                     |  |
| <code>Pathfinder(Graph G, String s)</code>           | 构造函数                |  |
| <code>int distanceTo(String v)</code>                | G 中从 s 到 v 的最短路径的长度 |  |
| <code>Iterable&lt;String&gt; pathTo(String v)</code> | G 中从 s 到 v 的最短路径    |  |

图的单源最短路径的 API

客户可以为给定的图 `G` 和源顶点（`source vertex`）`s` 构造一个 `PathFinder` 对象，然后使用该对象来查找最短路径的长度或者遍历从 `s` 到 `G` 中任何其他顶点的最短路径上的顶点。这些方法的一个实现被称为单源最短路径算法（`single-source shortest-path algorithm`）。我们将考虑一个解决这个问题的经典算法，称为广度优先搜索（`breadth-first search`），它提供了一个直接且简洁的解决方案。

**单源客户程序。**假设你已经有了一个廉价航空公司的地图，包含顶点和连接。然后，以你的家乡为源，你可以编写一个客户程序，在你想要去旅行的时候随时打印你的路线。程序 4.5.3 是 `PathFinder` 的一个客户程序，它为任意图提供这个功能。这种客户程序在处理由同

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
...
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine
```

```
% java IndexGraph amino.csv ","
TTA
 Lue
 L
 Leucine
Serine
 TCT
 TCC
 TCA
 TCG
```

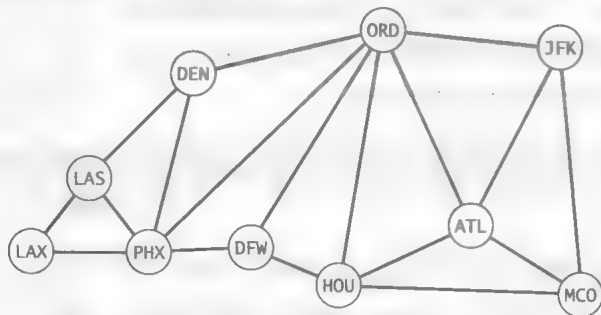
反转索引

682

683



一个源点出发，到不同终点的多个查询的应用中特别有用。在这种情况下，构建 PathFinder 对象的成本将在所有查询的成本上平摊。建议你通过在我们的示例输入文件 routes.txt 上运行 PathFinder 来探索最短路径的属性。



| 源点  | 终点  | 距离 | 一条最短路径              |
|-----|-----|----|---------------------|
| JFK | LAX | 3  | JFK-ORD-PHX-LAX     |
| LAS | MCO | 4  | LAS-PHX-DFW-HOU-MCO |
| HOU | JFK | 2  | HOU-ATL-JFK         |

图的最短路径的示例

程序4.5.3 最短路径客户程序

```
public class PathFinder
{
 // 程序4.5.3中有具体实现
 public static void main(String[] args)
 {
 // 读取图并计算源点s的最短路径
 In in = new In(args[0]);
 String delimiter = args[1];
 Graph G = new Graph(in, delimiter);
 String s = args[2];
 PathFinder pf = new PathFinder(G, s);

 // 处理查询
 while (StdIn.hasNextLine())
 {
 String t = StdIn.readLine();
 int d = pf.distanceTo(t);
 for (String v : pf.pathTo(t))
 StdOut.println(" " + v);
 StdOut.println("distance " + d);
 }
 }
}
```

|           |              |
|-----------|--------------|
| in        | 输入流          |
| delimiter | 输入分隔符        |
| G         | 图            |
| s         | 源点           |
| pf        | s的PathFinder |
| t         | 终点查询         |
| v         | 路径上的顶点       |

此PathFinder客户程序以文件名称、分隔符和源顶点作为命令行参数，它假定文件的每一行都指定了一个顶点和一个连接到该顶点的顶点列表（顶点之间用分隔符分隔），当你在标准输入中输入一个目的地时，你得到从源到目的地的最短路径。

```
% more routes.txt
JFK MCO
ORD DEN
PHX LAX
ORD HOU
DFW PHX
ORD DFW
...
JFK ORD
HOU MCO
LAS PHX
```

```
% java PathFinder routes.txt " " JFK
LAX
 JFK
 ORD
 PHX
 LAX
distance 3
HOU
 JFK
 ORD
 DFW
distance 2
```

分隔度。最短路径算法的经典应用之一是发现个人在社交网络中的分隔度 (degree of separation)。为了解决这个问题，我们用一个叫作凯文·贝肯的流行游戏来讨论这个应用，这个游戏使用了我们刚才讨论的电影-演员图。凯文·贝肯是一个高产的演员，出现在很多电影中。我们给在电影中出现过的每个演员分配一个凯文·贝肯数：贝肯本人是 0，与贝肯共演过的演员的凯文·贝肯数是 1；同这些数为 1 的演员，也就是这些与贝肯共演过的演员，再共演的其他任何演员（除了贝肯本人）的凯文·贝肯数是 2，以此类推。例如，梅丽尔·斯特里普的凯文·贝肯数是 1，因为她和凯文·贝肯出演了《狂野之河》；妮可·基德曼的数是 2：虽然她没有和凯文·贝肯一起出演过电影，但是她和唐纳德·萨瑟兰共同出演了《冷山》，而萨瑟兰和凯文·贝肯一起出演了《动物屋》。给出演员名字的情况下，这个

```
% java Pathfinder movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
 Bacon, Kevin
 Animal House (1978)
 Sutherland, Donald (I)
 Cold Mountain (2003)
 Kidman, Nicole
distance 4
Hanks, Tom
 Bacon, Kevin
 Apollo 13 (1995)
 Hanks, Tom
distance 2
```

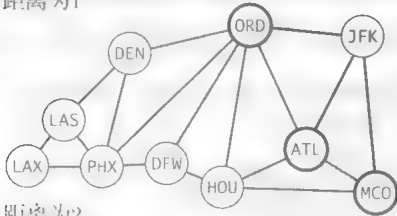
凯文·贝肯的分隔度

游戏的最简单的版本是找到一些交替的电影和演员的序列，可以指向凯文·贝肯。例如，一个电影迷可能知道汤姆·汉克斯曾经和劳埃德·布里奇斯共同出演了《魔岛仙踪》，而劳埃德·布里奇斯与帕特里克·艾伦共演了《电话谋杀案》，帕特里克·艾伦又与唐纳德·萨瑟兰共演了《猛鹰雄风》，唐纳德·萨瑟兰又与凯文·贝肯共演了《动物屋》。但是这个知识不足以建立汤姆·汉克斯的贝肯数（实际上他的贝肯数是 1，因为他和凯文·贝肯共演了《阿波罗 13 号》）。你可以看到，凯文·贝肯数其实就是计算最短序列的电影数，所以很难确定有人能够在不使用计算机的情况下赢得游戏。值得注意的是，程序 4.5.3 中的 Pathfinder 测试客户程序就是这样一个程序，你需要找到一个最短路径来计算 movies.txt 中任何演员的凯文·贝肯数——这个数是距离的一半（因为 Pathfinder 是路径长度，而游戏中只要电影数——译者注）。你可能会喜欢使用这个程序，或者扩展它来回答关于电影业务或许多其他领域的一些娱乐问题。例如，数学家们通过论文合著和与 20 世纪多产的数学家保罗·厄多斯的关系图来玩这个游戏。同样，新泽西州人的布鲁斯·斯普林斯汀数似乎都是 2，因为这个州的每个人似乎都认识一个自称知道布鲁斯的人。

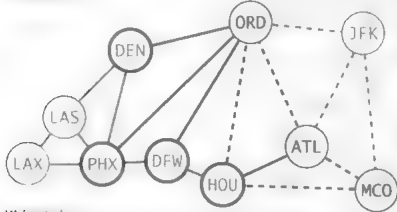
其他客户程序。PathFinder 是一种多用途的数据类型，可以有許多实际的用途。例如，通过为每个顶点构建一个 PathFinder，可以很容易地开发一个处理标准输入的任意源-目标请求的客户程序（参见练习 4.5.17）。旅行服务恰当使用这种方法可以非常高的服务率处理请求。由于该客户程序为每个顶点建立一个 PathFinder（每个顶点可能消耗与顶点数量成比例的内存），因此内存使用量可能是将其用于大型图的一个限制因素。对于设计原理相同但性能要求更高的应用，可以考虑一台互联网路由器，它具有可用机器之间的连接图，并且必须根据数据包给定的目的地，决定下一站如何传递数据才是最佳选择。要做到这一点，它可以建立一个以自己为源的 PathFinder；然后，为了发送一个数据包到目的地 w，它计算 pf.pathTo(w) 并将数据包发送到该路径上的第一个顶点——到 w 的最短路径的下一站。或者，中央权威机构可能会为多个相关路由器中的每一个建立一个 PathFinder 对象，并使用它们发布路由指令。如何以高服务率处理这些请求是互联网路由器的主要责任之一，而最短路径算法是该过程的关键部分。

为距离1进行初始化

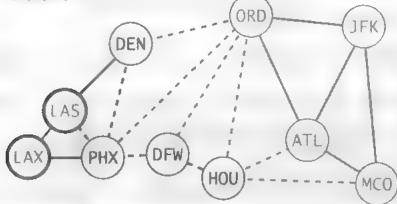
距离为1



距离为2



距离为3



为距离4进行检测

| v                   | 队列内容        | 距JFK的距离 |     |     |     |     |     |     |     |     |     |
|---------------------|-------------|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|                     |             | ATL     | DEN | DFW | HOU | JFK | LAS | LAX | MCO | ORD | PHX |
| JFK                 |             |         |     |     |     | 0   |     |     |     |     |     |
| JFK                 | ATL         | 1       |     |     |     | 0   |     |     |     |     |     |
| JFK                 | ATL MCO     | 1       |     |     |     | 0   |     | 1   |     |     |     |
| JFK                 | ATL MCO ORD | 1       |     |     |     | 0   |     | 1   | 1   |     |     |
| ATL MCO ORD         |             |         |     |     |     | 0   |     |     |     | 1   | 1   |
| ATL MCO ORD HOU     |             |         |     |     | 2   | 0   |     |     |     | 1   | 1   |
| MCO ORD HOU         |             |         |     |     | 2   | 0   |     |     |     | 1   | 1   |
| ORD HOU             |             |         |     |     | 2   | 0   |     |     |     | 1   | 1   |
| ORD HOU DEN         |             |         | 2   |     | 2   | 0   |     |     |     | 1   | 1   |
| ORD HOU DEN DFW     |             |         | 2   | 2   | 2   | 0   |     |     |     | 1   | 1   |
| ORD HOU DEN DFW PHX |             |         | 2   | 2   | 2   | 0   |     |     |     | 1   | 1   |
| HOU DEN DFW PHX     |             |         | 2   | 2   | 2   | 0   |     |     |     | 1   | 1   |
| DEN DFW PHX         |             |         | 2   | 2   | 2   | 0   |     |     |     | 1   | 1   |
| DEN DFW PHX LAS     |             |         | 2   | 2   | 2   | 0   | 3   |     |     | 1   | 1   |
| DFW PHX LAS         |             |         | 2   | 2   | 2   | 0   | 3   |     |     | 1   | 1   |
| PHX LAS             |             |         | 2   | 2   | 2   | 0   | 3   |     |     | 1   | 1   |
| PHX LAS LAX         |             |         | 2   | 2   | 2   | 0   | 3   | 3   |     | 1   | 1   |
| LAS LAX             |             |         | 2   | 2   | 2   | 0   | 3   | 3   |     | 1   | 1   |
| LAX                 |             |         | 2   | 2   | 2   | 0   | 3   | 3   |     | 1   | 1   |

#### 使用广度优先搜索来计算图中的最短路径距离

最短路径距离。理解广度优先搜索的第一步是考虑计算从源到其他顶点的最短路径长度的问题。我们的方法是计算并保存 PathFinder 构造函数中的所有距离，然后在客户程序调用 distanceTo() 时返回请求的值。要将整数距离与每个顶点名称关联起来，我们使用一个符号表：

```
ST<String, Integer> dist = new ST<String, Integer>();
```

这个符号表的目的是为每个顶点关联一个整数，这个整数是从 s 到该顶点的最短路径的长度（距离）。我们首先通过调用 dist.put(s,0) 把 s 的距离置为 0，然后使用以下代码将 s 的邻接顶点的距离设置为 1：

```
for (String v : G.adjacentTo(s))
 dist.put(v, 1)
```

但是，我们接下来该怎么做？如果我们盲目地将每个邻接顶点的所有邻接顶点的距离设置为 2，那么我们不仅会不必要地将许多值设置两次（邻接顶点可能有许多共同的邻接顶点），而且我们会把 s 的距离设置成 2（它是每个邻接顶点的邻接顶点），我们显然不想要这个结果。

解决这些困难的方法很简单：

- 按照与 s 的距离顺序考虑顶点。
- 忽略与 s 的距离已知的顶点。

为了组织计算，我们使用一个 FIFO 队列。从队列中的 s 开始，我们执行以下操作，直到队列为空：

- 顶点  $v$  出队列。
- 给  $v$  的所有未知的邻接顶点设置距离，其值比  $v$  的距离大 1。
- 所有未知的邻接顶点入队列。

广度优先搜索使这些顶点以到源  $s$  的距离非降序排列。在示例图上追踪这个算法将有助于说服你它是正确的。如何证明广度优先搜索标记出每个顶点  $v$  到  $s$  的距离的过程是正确的，我们把它留作数学归纳的一个练习（见练习 4.5.12）。

**最短路径树。**我们不仅要求最短路径的长度，还要求最短路径本身。为了实现 `pathTo()`，我们使用一个称为最短路径树（short-paths tree）的子图，定义如下：

- 将源放在树的根上。
- 处理顶点  $v$  时，如果  $v$  的邻接顶点入队列，那么把它们放到树中，并且每个邻接顶点都用一条边连接到  $v$ 。

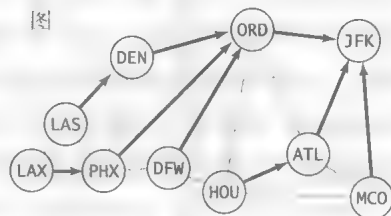
由于我们只将每个顶点入队列一次，所以这个结构显然会是一棵树：它由根（源）连接到源的每个邻接顶点的子树组成。研究这样的树，你可以立即知道树中每个顶点到根的距离与图中源的最短路径的长度相同。更重要的是，树中的每条路径都是图中的最短路径。这一观察非常重要，因为它使我们能够轻松地为客户提供最短路径。首先，我们维护一个符号表，它将每个顶点与它所在的最短路径上的前一个顶点（即与源点距离更近一步的顶点）相关联：

```
ST<String, String> prev;
prev = new ST<String, String>();
```

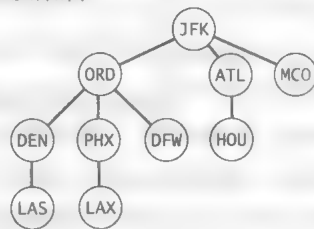
对于每个顶点  $w$ ，我们希望把它与从源到  $w$  的最短路径上的前一个顶点关联起来。通过广度优先搜索来计算这个信息是很容易的：当我们发现顶点  $w$  是  $v$  的邻接顶点时我们把  $w$  入队列，我们这样做是因为  $v$  是从源节点到  $w$  的最短路径的上一个节点，所以我们可以调用 `prev.put(w,v)` 来记录这些信息。`prev` 数据结构不过是最短路径树的一种表示：它提供了从树中每个节点到其父节点的链接。然后，为了响应客户程序从源到  $v$  的最短路径的请求，我们沿着这个从  $v$  开始的树，按照相反的顺序遍历路径，所以我们把遇到的每个顶点推到一个栈上，然后给客户程序返回这个栈（它是 `Iterable` 型）。栈顶是源  $s$ ；栈底是  $v$ ；并且从  $s$  到  $v$  的路径上的顶点在两者之间，所以当使用 `foreach` 语句中的 `pathTo()` 返回值时，客户程序得到的就是从  $s$  到  $v$  的路径。

**广度优先搜索。**`PathFinder`（程序 4.5.4）是基于刚

刚讨论的思想的单源最短路径 API 的实现。它使用了两个符号表：一个用于从源到每个顶点的距离，另一个用于从源到每个顶点的最短路径上的前一个点。构造函数使用一个 FIFO 队列追踪已经遇到的顶点（对于那些已经确定其最短路径但其邻接顶点尚未被检查的顶点，队列中保存了它们的邻接顶点）。这个过程被称为广度优先搜索（breadth-first search, BFS），因为它在图中按层进行了尽可能增大宽度的搜索。相比之下，还有一种重要的图搜索方法被称为深度优先搜索（depth-first search），它是一种基于递归的方法，就像我们在程序 2.4.5 中用于处理渗透的那种递归方法，这种方法是尽可能深入地对图进行搜索。深度优先搜索倾向于



最短路径树



父链接表示

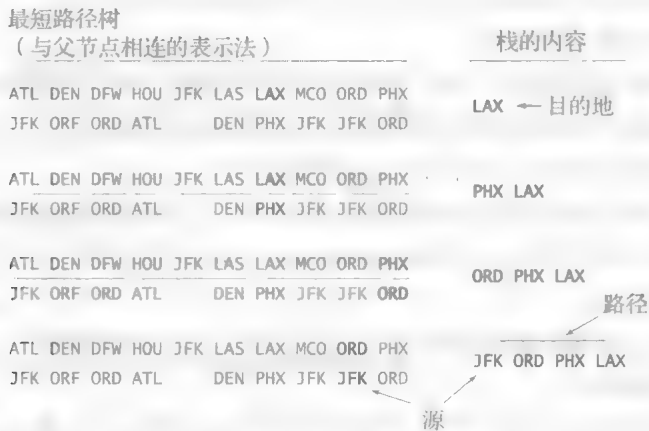
|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| ATL | DEN | DFW | HOU | JFK | LAS | LAX | MCO | ORD | PHX |
| JFK | ORD | ORD | ATL |     | DEN | PHX | JFK | JFK | ORD |

最短路径树

689

690

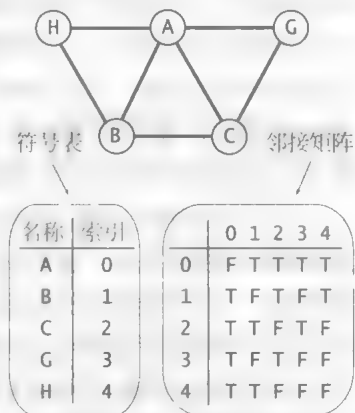
找到长路径；广度优先搜索可以保证找到最短路径。



使用栈从最短路径树中恢复路径

性能。图处理算法的成本通常取决于两个图参数：顶点数  $V$  和边数  $E$ 。如在 PathFinder 中所实现的，宽度优先搜索所需的时间与输入大小呈线性关系，在最坏的情况下与  $E \log V$  成正比。为了证明这一点，首先要注意的是，外循环（while）至多迭代  $V$  次，每个顶点一次，因为我们要确保每个顶点最多入队一次；然后可以观察到内循环（for）在所有迭代中总共迭代至多  $2E$  次，因为我们要确保每个边最多检查两次，连接每个边的两个节点各一次。在大小不超过  $V$  的符号表上，循环的每次迭代至少需要一个 contains() 操作，以及可能两个 put() 操作。这个线性对数时间性能取决于使用基于二叉搜索树的符号表（比如 ST 或 java.util.TreeMap），它既有线性时间的搜索，也有对数时间的搜索和插入。使用散列表（如 java.util.HashMap）来替换符号表能够将输入大小的运行时间减少为线性，与典型图的  $E$  成比例。

邻接矩阵表示。如果没有适当的数据结构，图处理算法有时无法得到高性能的实现，所以不应视其为理所当然。例如，图还有一种表示方法称为邻接矩阵表示（adjacency-matrix representation），使用符号表将顶点名称映射到 0 到  $V-1$ <sup>①</sup> 之间的整数，然后维护一个  $V \times V$  的布尔数组，如果存在一个连接顶点  $i$  和顶点  $j$  的边，那么数组  $i$  行  $j$  列的元素就是 true，如果不存在这条边，则为 false。在本书中，我们已经使用了类似的表示方法，如在 1.6 节中研究用于网页排序的随机游走模型。邻接矩阵表示法很简单，但对于大型图来说是不可行的，一个具有百万个顶点的图需要一个具有万亿个元素的邻接矩阵。理解图处理问题的这种区别，决定了最终是否能够解决一个实际问题。



图的邻接矩阵表示

广度优先搜索是一种基本的算法，你可以使用这种算法在航空公司路线图、城市地铁系统（参见练习 4.5.38）或许多类似的情况中找到路线。正如我们在分隔度的例子中所表明的那样，它也被用于无数的其他应用，从互联网上的网页、路由数据包到研究传染病、大脑模型以及基因组序列之间的关系。这些应用许多都涉及巨大的图，所以一个高效的算法是必不可少的。

① 原书是  $V-1$ ，出现得很突兀，估计是笔误。——译者注

程序4.5.4 最短路径实现

```
public class Pathfinder
{
 private ST<String, Integer> dist;
 private ST<String, String> prev;

 public Pathfinder(Graph G, String s)
 { // 使用BFS计算源顶点s到图G中的每个其他顶点
 的最短路径
 prev = new ST<String, String>();
 dist = new ST<String, Integer>();
 Queue<String> queue = new Queue<String>();
 queue.enqueue(s);
 dist.put(s, 0);
 while (!queue.isEmpty())
 { // 处理队列中的下一个顶点
 String v = queue.dequeue();
 for (String w : G.adjacentTo(v))
 { // 检查距离是否已知
 if (!dist.contains(w))
 { // 添加到队列中; 保存最短路径信息
 queue.enqueue(w);
 dist.put(w, 1 + dist.get(v));
 prev.put(w, v);
 }
 }
 }
 }

 public int distanceTo(String v)
 { return dist.get(v); }

 public Iterable<String> pathTo(String v)
 { // 从s到v的最短路径上的顶点
 Stack<String> path = new Stack<String>();
 while (v != null && dist.contains(v))
 { // 当前顶点入队; 移动到路径上的前一个顶点
 path.push(v);
 v = prev.get(v);
 }
 return path;
 }
}
```

|      |                 |
|------|-----------------|
| dist | 距s的距离           |
| prev | 从s开始的最短路径的前一个节点 |

|   |        |
|---|--------|
| G | 图      |
| s | 源      |
| q | 顶点队列   |
| v | 当前顶点   |
| w | v的邻接顶点 |

|              |          |
|--------------|----------|
| PathFinder() | G中s的构造函数 |
| distanceTo() | 从s到v的距离  |
| pathTo()     | 从s到v的路径  |

这个类使用广度优先搜索来计算从特定源顶点s到图G中每个顶点的最短路径。请参阅程序4.5.3中的示例客户程序。

最短路径问题的一个重要推广是为每个边设置权重（可以表示距离或时间），并寻求找到边的权重总和最小化的路径。如果你接下来学习算法或者运筹学，你将会学到广度优先搜索的一个通用算法——Dijkstra 算法，它可以在线性时间内解决这个问题。当你从 GPS 设备或网络上的地图应用程序获取路线时，Dijkstra 算法是解决相关最短路径问题的基础。这些重要并且常见的应用程序只是冰山一角，因为图模型比地图要普遍得多。

**小世界图** 科学家已经确定了一类特别有趣的图，被称为小世界图（small-world graphs），它出现在自然科学和社会科学的众多应用中。小世界图由以下三个属性表征：

- 稀疏性：边的数量远小于图中顶点数可以对应的边的总数（图中任意两个顶点就能对应一条边，因此顶点数决定了整个图中可以包含的边数的上限，稀疏图就是指图中存在的边数远小于这个上限的图——译者注）。
- 平均路径长度较短：如果选取两个随机顶点，则它们之间的最短路径的长度很短。
- 它们呈现局部聚集：如果有两个顶点同时是同一个顶点的邻接顶点，那么这两个顶点很可能是彼此的邻接顶点。

我们将具有这三种属性的图统称为呈现了小世界现象 (small world phenomenon)。“小世界”这个术语指的是多数顶点既有局部聚集,又有到其他顶点的短路径。“小世界现象”这个词指出这样一个意想不到的事实,即在实践中出现的很多图都是稀疏的,呈现局部聚集,并且路径很短。除了刚刚考虑的社交关系应用之外,小世界图已被用于研究产品或想法的营销、时尚的形成和传播、互联网的分析、安全的端对端网络的构建、路由算法和无线网络的发展、电网的设计、人脑信息处理的建模、振荡器中相变的研究、病毒感染(在生物体和计算机中)的传播等应用。从20世纪90年代沃茨和斯特罗加茨的开创性工作开始,人们对小世界现象进行了大量的研究。

这样的研究中的一个关键问题如下:给出一个图,我们如何判断它是否是一个小世界图?为了回答这个问题,我们首先增加一个条件,就是这个图是不小的(比如,它有1000个顶点或更多),并且它是连通的(存在连接每对顶点的路径)。然后,我们需要为每个小世界属性设置具体的阈值:

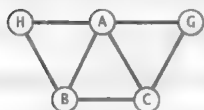
- 稀疏性是指平均顶点的度小于  $20\lg V$ 。
- 平均路径长度较短是指两个顶点之间最短路径的平均长度小于  $10\lg V$ 。
- 对于局部聚集,我们规定一个术语聚集系数 (clustering coefficient),小世界的聚集系数数值应大于 10%。

693

局部聚集的定义比稀疏性和平均路径长度的定义稍微复杂一些。直观地说,顶点的聚集系数表示的是:随机选取一个顶点的两个邻接顶点,这两者之间被一条边连接起来的概率。更确切地说,如果一个顶点有  $t$  个邻接顶点,那么有  $t(t-1)/2$  条边连接这些邻接顶点,局部聚集系数指的就是这些边所占比例。当顶点度为 1 或 0 时,其局部聚集系数为 0。图的聚集系数 (clustering coefficient of a graph) 是其顶点的局部聚集系数的平均值。如果这个平均值大于 10%,我们就说这个图形是局部聚集的。下图计算了一个小型图的这三个值。

平均顶点度数

| 顶点   | 度                      |
|------|------------------------|
| A    | 4                      |
| B    | 3                      |
| C    | 3                      |
| G    | 2                      |
| H    | 2                      |
| 总数   | 14                     |
| 平均度数 | $= \frac{14}{5} = 2.8$ |



平均路径长度

| 顶点对 | 最短路径  | 长度 |
|-----|-------|----|
| A B | A-B   | 1  |
| A C | A-C   | 1  |
| A G | A-G   | 1  |
| A H | A-H   | 1  |
| B C | B-C   | 1  |
| B G | B-A-G | 2  |
| B H | B-A-H | 2  |
| C G | C-A-G | 2  |
| C H | C-A-H | 2  |
| G H | G-A-H | 2  |
| 总数  |       | 13 |

聚集系数

|    |   | 连接邻接顶点的边                                              |     |
|----|---|-------------------------------------------------------|-----|
| 顶点 | 度 | 实际数                                                   | 可能数 |
| A  | 4 | 3                                                     | 6   |
| B  | 3 | 2                                                     | 3   |
| C  | 3 | 2                                                     | 3   |
| G  | 2 | 1                                                     | 1   |
| H  | 2 | 1                                                     | 1   |
|    |   | $\frac{3/6 + 2/3 + 2/3 + 1/1 + 1/1}{5} \approx 0.767$ |     |

$$\frac{\text{长度总数}}{\text{顶点对的个数}} = \frac{13}{10} = 1.3$$

#### 计算小世界图的特征

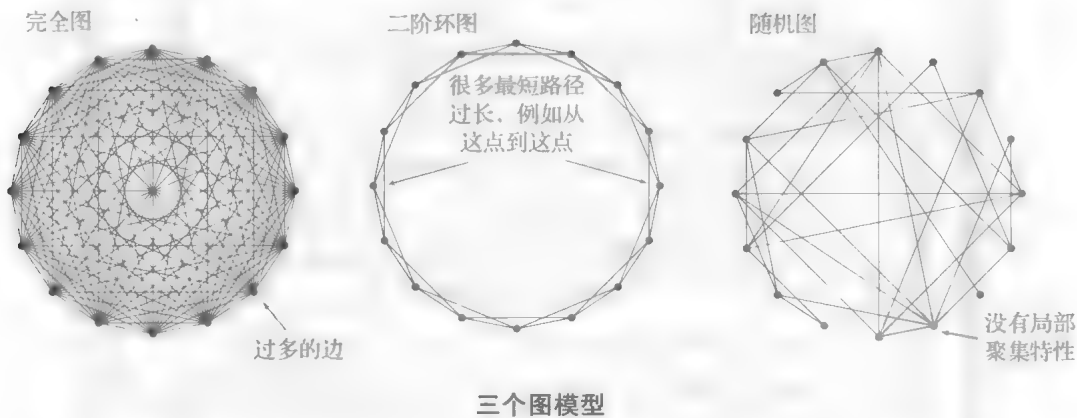
为了更好地熟悉这些定义,我们接下来定义一些简单的图模型,并检查它们是否具备这三个必要的属性以描述小世界图。

完全图。具有  $V$  个顶点的完全图 (complete graph) 具有  $V(V-1)/2$  条边,每条边连接每对顶点。完全图不是小世界图。它们的平均路径长度较短(每条最短路径长度为 1),它们表现出局部聚集(聚集系数为 1),但它们并不稀疏(平均顶点度数为  $V-1$ ,对于较大的  $V$ ,远大于  $20\lg V$ )。



轮环图。轮环图 (ring graph) 是  $V$  个顶点均匀地分布在一个圆的圆周上, 每个顶点都连接到其两侧的邻接顶点。在  $k$  阶环图 ( $k$ -ring graph) 中, 每个顶点都与其两边的  $k$  个最近邻接顶点邻接。下图展示了一个有 16 个顶点的二阶环图。轮环图也不是小世界图。例如, 二阶环图是稀疏的 (每个顶点具有度数 4) 并且是局部聚集的 (聚集系数是  $1/2$ ), 但是它们的平均路径长度并不短 (参见练习 4.5.20)。

随机图。Erdős-Renyi 模型是生成随机图 (random graph) 的一个模型, 已经得到深入的研究。在这个模型中, 我们通过概率为  $P$  的边连接  $V$  个顶点以建立随机图。具有足够数量的边的随机图看起来很像是连通的, 并且平均路径长度较短, 但它们不是小世界图, 因为它们不是局部聚集的 (参见练习 4.5.46)。



这些例子说明, 开发同时满足所有三个属性的图模型是一个令人困惑的挑战。花点时间尝试设计一个你认为可以满足的图模型。在思考这个问题之后, 你会意识到你可能需要一个程序来帮助计算。另外, 你可能也意识到, 在实践中能够经常发现这样的图是相当令人惊讶的。事实上, 你可能也想知道任意的一个图是不是小世界图!

| 模型   | 是否稀疏 | 最短路径是否较短 | 是否局部聚类 |
|------|------|----------|--------|
| 完全图  | ○    | ●        | ●      |
| 二阶环图 | ●    | ○        | ●      |
| 随机图  | ●    | ●        | ○      |

图模型的小世界属性

其实我们的限定条件是有一些主观成分的, 如聚集阈值选择 10% 而不是某个其他固定的百分比, 以及稀疏性阈值选择  $20\lg V$ 、短路径阈值选择  $10\lg V$  等, 但是我们通常不会接近这些边界值。例如, 考虑网页图 (web graph), 其中每个网页都是一个顶点, 并且如果两个页面之间有链接, 那就用一条边连接这两个顶点。科学家估计, 从一个网页到达另一个网页需要的点击次数很少会超过 30。由于网页数量达到数十亿, 这个估计意味着平均路径长度非常短, 远低于我们的  $10\lg V$  阈值 (对于十亿个顶点, 这将约为 300)。

在定义之后, 测试图是不是小世界图仍然是一个重要的计算负担。像你可能怀疑过的那样, 我们一直在考虑的图处理数据类型正好可以用来开发我们需要的工具。SmallWorld (程序 4.5.5) 是一个 Graph 和 PathFinder 的客户程序, 实现了这些测试。如果没有我们已经考虑过的有效的数据结构和算法, 这个计算的代价就会过高。即便如此, 对于大型图 (如 movies.txt), 我们也必须借助于统计抽样, 才能在合理的时间内估计平均路径长度和聚集系数 (参见练习 4.5.44), 因为函数 averagePathLength() 和 clusteringCoefficient() 需要平方量级时间。

程序4.5.5 小世界测试

```

public class SmallWorld
{
 public static double averageDegree(Graph G)
 { return 2.0 * G.E() / G.V(); }

 public static double averagePathLength(Graph G)
 { // 计算平均顶点距离
 int sum = 0;
 for (String v : G.vertices())
 { // 加入到距v的边长的总和中
 Pathfinder pf = new Pathfinder(G, v);
 for (String w : G.vertices())
 sum += pf.distanceTo(w);
 }
 return (double) sum / (G.V() * (G.V() - 1));
 }

 public static double clusteringCoefficient(Graph G)
 { // 计算聚集系数
 int total = 0;
 for (String v : G.vertices())
 { // 累积顶点v的局部聚集系数
 int possible = G.degree(v) * (G.degree(v) - 1);
 int actual = 0;
 for (String u : G.adjacentTo(v))
 for (String w : G.adjacentTo(v))
 if (G.hasEdge(u, w)) actual++;
 total += 1.0 * actual / possible;
 }
 return (double) total / G.V();
 }

 public static void main(String[] args)
 { /* 参见练习4.5.24 */ }
}

```

图  
顶点之间距离的累加和  
顶点迭代器变量  
v的邻接顶点

图  
可能的局部边的累积总和  
实际局部边的累积总和  
顶点迭代器变量  
v的邻接顶点

这个客户程序从标准输入读取一个图，并计算图的各种参数的值，以测试该图是否呈现出小世界现象。

```

% java SmallWorld "/" tinyGraph.txt
5 vertices, 7 edges
average degree = 2.800
average path length = 1.300
clustering coefficient = 0.767

```

一个经典的小世界图。我们的电影-演员图不是一个小世界图，因为它是个二部图，因此聚集系数为0。另外，一些演员之间没有任何路径相互连接。然而，如果两个演员同时出现在同一部电影中，则将其连接在一起，这样就可以得到一个更简单的演员-演员图，它就是一个典型的小世界图（在丢弃那些没有连接到凯文·贝肯的演员后）。下图说明了与一个小型的电影-演员列表文件相对应的电影-演员图和演员-演员图。

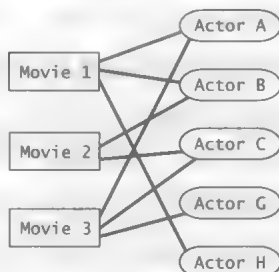
电影-演员列表文件

```

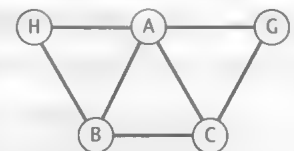
% more tinyMovies.txt
Movie 1/Actor A/Actor B/Actor H
Movie 2/Actor B/Actor C
Movie 3/Actor A/Actor C/Actor G

```

电影-演员图



演员-演员图



电影-演员列表文件的两种不同的图表示

Performer (程序 4.5.6) 是一个从电影 - 演员列表输入格式的文件创建演员 - 演员图的程序。电影 - 演员列表文件中的每一行由一个电影组成, 后面跟着所有出现在该电影中的演员, 由斜线分隔。Performer 增加了连接出现在该电影中的每对演员的边。对输入中的每个电影执行此操作会根据需要生成连接演员的图。

696  
697

程序4.5.6 演员-演员图

```
public class Performer
{
 public static void main(String[] args)
 {
 String filename = args[0];
 String delimiter = args[1];
 Graph G = new Graph();
 In in = new In(filename);
 while (in.hasNextLine())
 {
 String line = in.readLine();
 String[] names = line.split(delimiter);
 for (int i = 1; i < names.length; i++)
 for (int j = i+1; j < names.length; j++)
 G.addEdge(names[i], names[j]);
 }
 double degree = SmallWorld.averageDegree(G);
 double length = SmallWorld.averagePathLength(G);
 double cluster = SmallWorld.clusteringCoefficient(G);
 StdOut.printf("number of vertices = %7d\n", G.V());
 StdOut.printf("average degree = %7.3f\n", degree);
 StdOut.printf("average path length = %7.3f\n", length);
 StdOut.printf("clustering coefficient = %7.3f\n", cluster);
 }
}
```

|         |              |
|---------|--------------|
| G       | 图            |
| in      | 输入流          |
| line    | 电影-演员列表文件的一行 |
| names[] | 电影和演员名称      |
| i, j    | 两个演员的索引      |

该程序是一个SmallWorld的客户程序, 它将电影-演员列表的文件名和分隔符作为命令行参数, 并创建关联的演员-演员图。它将图的顶点数、平均度数、平均路径长度和聚集系数打印到标准输出。它假定演员-演员图是连通的 (参见练习4.5.29), 以便定义平均路径长度<sup>⊖</sup>。

```
% java Performer tinyMovies.txt "/"
number of vertices = 5
average degree = 2.800
average path length = 1.300
clustering coefficient = 0.767
```

```
% java Performer moviesG.txt "/"
number of vertices = 19044
average degree = 148.688
average path length = 3.494
clustering coefficient = 0.911
```

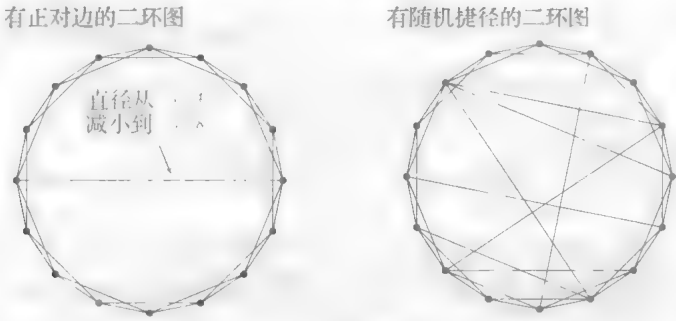
698

由于演员 - 演员图通常具有比相应的电影 - 演员图更多的边, 所以我们将使用来自文件 moviesG.txt 的小型演员 - 演员图来进行处理, 其包含 1261 个 G 级电影和 19 044 个演员 (全部连接到凯文·贝肯)。现在, Performer 告诉我们, 与 moviesG.txt 相关的演员 - 演员图有 19 044 个顶点和 1 415 808 条边, 所以平均顶点度数为 148.7 (大约是  $20 \lg V = 284.3$  的一半), 这意味着它是稀疏的; 其平均路径长度为 3.494 (远低于  $10 \lg V = 142.2$ ), 所以路径较短; 其聚集系数为 0.911, 因此具有局部聚集。我们找到了一个小世界图! 这些计算验证了这种社会关系图可以表现出小世界现象。鼓励你查找其他真实世界的图, 并使用 SmallWorld 对其进行测试。

⊖ 原书上是 average page length, 感觉像是笔误。——译者注

一种理解小世界现象的方法是开发一个数学模型，我们可以用它来检验假设和做出预测。最后，我们回到如何生成一个图模型的问题上，这个模型可以帮助我们更好地理解小世界现象。找到一个这种模型的取巧方法是组合两个稀疏图：一个二阶轮环图（具有较高的聚集系数）和一个随机图（具有较小的平均路径长度）。

具有随机捷径的轮环图。沃茨和斯特罗加茨的工作中揭示的最令人惊讶的事实之一是，将相对少量的随机边添加到具有局部聚类的稀疏图中会产生一个小世界图。为了深入了解为什么会出现这种情况，可以考虑一个二阶轮环图，其中直径（最远的一对顶点之间的路径长度）为 $\sim V/4$ （见下图）。增加一个连接最远顶点的边后，直径减小到 $\sim V/8$ （见练习 4.5.21）。将  $V/2$  个随机“捷径”边添加到二阶环图中极有可能显著降低平均路径长度，使其成为对数级（请参见练习 4.5.25）。而且，这样做将平均度数仅提高了 1，并且没有将聚集系数降低到远低于  $1/2$ 。也就是说，具有  $V/2$  条“捷径”的二阶环图很可能是一个小世界图！



一个新的图模型

依据这些模型创建图的生成器是很容易开发的，我们可以使用 SmallWorld 来确定图是否展现出小世界现象（见练习 4.5.24）。我们还可以验证我们为 tinyGraph.txt 的简单图、完全图或轮环图导出的分析结果。然而，与大多数科学研究一样，我们在回答旧的问题的同时出现了新的问题。我们需要添加多少条随机“捷径”才能获得较短的平均路径长度？随机连接图中的平均路径长度和聚集系数是多少？还有哪些其他的图模型可能适合于研究？精确地估计一个大型图中的聚集系数或者平均路径长度需要多少样本？你可以在练习中找到很多解决这些问题的建议，并进一步调研小世界现象。借助本书开发的基本工具和编程方法，你可以很好地解决这些以及很多其他的科学问题。

**经验总结** 这个案例研究说明了算法和数据结构在科学研究中的重要性。这也强化了我们在本书中学到的一些值得重复说明的经验教训。

**仔细设计你的数据类型。**在本书中，我们长久坚持的理念之一就是，有效的编程来自于对数据类型可能取值的集合以及在这些值上定义的操作集合的精确理解。使用像 Java 这样的面向对象的现代编程语言为我们提供了理解的途径，因为我们设计、构建和使用我们自己的数据类型。我们的 Graph 数据类型是一个基本的数据类型，是很多迭代以及我们讨论过的设计经验的产物。我们客

| 模型                    | 平均度数     | 平均路径长度      | 聚集系数       |
|-----------------------|----------|-------------|------------|
| 完全图                   | 999<br>○ | 1<br>●      | 1.0<br>●   |
| 二阶环图                  | 4<br>●   | 125.38<br>○ | 0.5<br>●   |
| $p = 10/1$ 的随机连接图     | 10<br>●  | 3.26<br>●   | 0.010<br>○ |
| 具有 $V/2$ 条随机“捷径”的二阶环图 | 5<br>●   | 5.71<br>●   | 0.343<br>● |

具有 1000 个顶点的不同种类的图的小世界参数

户程序代码清晰和简单，证明了在任何程序中认真对待基本数据类型的设计和实现的价值。 [700]

逐步开发代码。与我们所有其他案例研究一样，我们一次构建一个软件模块，在转到下一个模块之前测试和学习每个模块。

在解决未知问题之前解决你能理解的问题。我们在几个城市之间设计航线的最短路线的示例是一个很容易理解的简单示例，但又足够复杂到能够引起我们的兴趣，以便坚持调试和追踪路径，又不会复杂到使这些任务变得太繁重。

持续测试并检查结果。当调用处理大量数据的复杂程序时，你一定要仔细地检查结果，并使用常识来评估程序产生的每一个输出。新手程序员会怀有乐观的心态（“如果程序产生了答案，它肯定是正确的”）；有经验的程序员却怀揣着悲观的心态（“这个结果肯定有什么问题”）。

使用真实世界的的数据。网络电影数据库（Internet Movie Database）中的 movies.txt 文件只是数据文件的一个例子，互联网上存在大量这样的数据。在过去，这样的数据经常被隐藏为私用或专有格式，但是如今大多数人意识到简单的文本格式才是首选。Java 的 String 数据类型中的各种方法使得处理真实数据变得容易，因此也可以用来模拟现实世界中的各种现象。刚开始使用时，建议使用真实世界格式的小型文件，以便在处理大型文件之前测试程序的正确性，并获得性能信息。

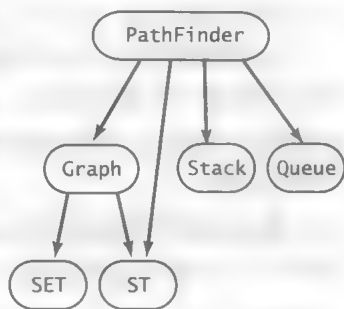
软件复用。本书中我们持久坚持的另一个理念是，有效的编程基于对我们可用的基本数据类型的理解，所以我们不必为了基本功能而重写代码。我们在 Graph 中使用 ST 和 SET 是一个很好的例子——很多程序员仍然使用图的低级表示和实现，这些实现方法使用链表或数组表示图，这意味着他们必须为任何简单的操作都要再写一遍代码，比如维护和遍历链表。我们的最短路径类 Pathfinder 使用了 Graph、ST、SET、Stack 和 Queue——囊括了几乎所有的基本数据结构。

保持灵活性。重复使用软件通常意味着使用各种 Java 库中的类。这些类的接口通常非常宽泛（即它们包含很多方法），因此，即使你的实现都是 Java 库方法的调用，定义和实现自己的 API 以及定义客户程序和实现之间的狭窄接口总是明智的。这种方法提供了灵活性，这样你就可以在有保证的情况下切换到更有效的实现，并避免依赖对库中自己不使用的部分的更改。例如，在我们的 Graph 实现（程序 4.5.1）中使用 ST 使我们能够灵活地使用我们的任何符号表实现（如 HashST 或 BST），或是可以灵活使用 Java 的符号表实现（java.util.TreeMap 和 java.util.HashMap），而不必更改 Graph。

性能影响。如果没有好的算法和数据结构，本章的许多问题就无法被解决，因为很多初级的简单方法需要的时间和空间消耗是难以忍受的。估算我们程序的资源需求是设计中非常重要的一环。

这个案例研究是本章的完美收官，因为它很好地说明了我们所考虑的方案是一个起点，而不是一个完整的研究。到目前为止，我们已经介绍的编程技巧也是一个起点，对于你在科学、数学、工程学或任何计算起着重要作用的研究领域（现在几乎是任何领域）的深入研究而言，都是一个起点。编程方法和你在这里学到的工具应该为你解决任何计算问题做好了准备。

在对现代编程语言有了了解和信心后，你肯定已准备好独立地去思考和计算。这些可以把你引入计算的新境界，就是在未来，无论你遇到什么样的计算问题，它们都一定会对你很



Pathfinder 的代码复用

**[702]** 有帮助。接下来，我们开始这个旅程。

### 问答环节

**问：**给定  $V$  个顶点，请问存在有多少个不同的图？

**答：**假设没有自环或平行边，就有  $V(V-1)/2$  条可能的边，每条边可以存在或不存在，所以总和为  $2^{V(V-1)/2}$ 。这个数值呈指数级增长，如下表所示：

| $V$            | 1 | 2 | 3 | 4  | 5    | 6      | 7         | 8           | 9              |
|----------------|---|---|---|----|------|--------|-----------|-------------|----------------|
| $2^{V(V-1)/2}$ | 1 | 2 | 8 | 64 | 1024 | 32 768 | 2 097 152 | 268 435 456 | 68 719 476 736 |

这些巨大的数字表明了社会关系的复杂性。例如，在街上，你只考虑从此刻开始你遇见的 9 个人，他们相互认识的可能性就超过 68 万亿！

**问：**一个图中是否可以有一个顶点不与图中的任何顶点相邻？

**答：**好问题。这样的顶点被称为孤立顶点 (isolated vertices)。我们的实现中是不允许它们存在的。其他实现若提供了通过 `addvertex()` 方法来实现显式地添加一个顶点的操作，则允许孤立顶点的存在。

**问：**为什么不对每个顶点的邻接顶点只使用链表的表示？

**答：**你可以这样做，但是当你发现需要大小、迭代器等时，你很可能会重新实现基本链表的代码。

**问：**为什么 `V()` 和 `E()` 查询方法需要用常数时间实现？

**答：**或许大多数客户程序只会调用一次这样的方法，但是也可能出现如下代码：

```
for (int i = 0; i < G.E(); i++)
{ ... }
```

若你偷懒使用一个算法来计算边的数量而非通过维护一个实例变量来计算边的数量，你将需要花费平方级的时间。详见练习 4.5.1。

**[703]**

**问：**为什么 `Graph` 和 `PathFinder` 在不同的类中？在 `Graph` API 中包含 `PathFinder` 方法是否更有意义？

**答：**找到最短路径只是众多图处理问题之一。如果将所有这些都包含在一个 API 中，这将是—个糟糕的软件设计。请重新阅读 3.3 节关于宽接口的讨论。

**[704]**

### 练习

**4.5.1** 把返回图中顶点和边的数量的 `V()` 和 `E()` 实现分别添加到 `Graph` 中。确保你的实现需要常数时间。提示：对于 `V()`，你可能会假设 `ST` 中的 `size()` 方法需要常数时间；对于 `E()`，维护一个实例变量来保存图中边的当前数量。

**4.5.2** 给 `Graph` 添加一个方法 `degree()`，它接收一个字符串参数并返回指定顶点的度数。使用此方法可查找文件 `movies.txt` 中哪个演员在电影中出现得最多。

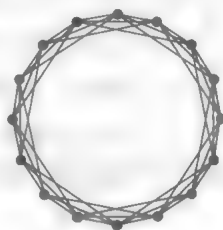
答案：

```
public int degree(String v)
{
 if (st.contains(v)) return st.get(v).size();
 else return 0;
}
```

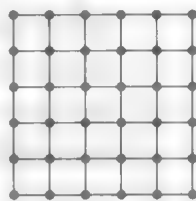
- 4.5.3 为 Graph 添加一个方法 `hasVertex()`，它接收一个字符串参数，如果这个字符串与图中顶点名称相同，则返回 `true`，否则返回 `false`。
- 4.5.4 为 Graph 添加一个方法 `hasEdge()`，它接收两个字符串参数，如果它们在图中确定了边，则返回 `true`，否则返回 `false`。
- 4.5.5 为 Graph 创建一个拷贝构造函数，将图 G 作为参数，然后创建并初始化一个新的独立副本。图 G 将来的任何变化都不会影响新创建的图。
- 4.5.6 编写支持显式顶点创建的 Graph 版本，并允许自环、平行边和孤立顶点。提示：为邻接顶点列表使用 `Queue` 而不是 `SET`。
- 4.5.7 为 Graph 添加一个 `remove()` 方法，该方法接收两个字符串参数，并从图中删除指定的边（如果存在的话）。
- 4.5.8 给图添加一个方法 `subgraph()`，它以 `SET<String>` 为参数，并返回导出子图（该图包含了所有指定顶点，以及原图中两个顶点都在指定顶点集中的所有的边）。 705
- 4.5.9 编写一个支持可比较的泛型顶点类型的 Graph 版本（简单任务）。然后，编写一个 `PathFinder` 版本，支持使用你编写的可比较的泛型顶点类型来查找最短路径（困难任务）。
- 4.5.10 创建上一个练习中 Graph 的一个版本，以支持二部图（图中所有的边都是从一类可比较的泛型顶点连接到另一类可比较的泛型顶点）。
- 4.5.11 判断题：在广度优先搜索期间的某时刻，队列可以包含两个顶点，一个与源距离为 7，一个距离为 9。  
答案：错。该队列可以包含至多两个不同距离  $d$  和  $d+1$  的顶点。广度优先搜索以距离源的距离递增的顺序来检查顶点。当检查距离为  $d$  的顶点时，只有距离  $d+1$  的顶点可以入队。
- 4.5.12 通过归纳法证明 `PathFinder` 可以计算出从源到每个顶点的最短路径（和最短路径距离）。
- 4.5.13 假设你在 `PathFinder` 中使用栈而非队列进行广度优先搜索。它是否仍然可以计算从源到每个顶点的路径？它是否仍然可以计算最短路径？分别证明或给出一个反例。
- 4.5.14 在 `pathTo()` 中生成最短路径时，使用队列而不是栈的效果是什么？
- 4.5.15 向 `PathFinder` 添加 `isReachable(v)` 方法，如果从 `source` 到 `v` 存在路径，返回 `true`，否则返回 `false`。
- 4.5.16 编写一个 Graph 客户程序，用于从文件中读取图（以正文中指定的文件格式），然后打印图中的边，每行一个。
- 4.5.17 实现一个 `PathFinder` 的客户程序 `AllShortestPaths`，它为每个顶点创建一个 `PathFinder` 对象，以及一个测试客户程序，它从标准输入中获得一对顶点信息，用于查询并打印连接它们的最短路径。支持分隔符，以便你可以在一行中输入两个字符串以查询（用分隔符分隔），并输出它们之间的最短路径。注意：对于 `movies.txt`，查询字符串可能两个都是演员，可能两个都是电影，也可以是一个演员和一部电影。 706
- 4.5.18 编写一个程序，在一个包含 1000 个顶点的二阶环图中，增加若干条随机捷径，请绘制平均路径长度和随机捷径边的数量之间的关系图。
- 4.5.19 重载 `SmallWorld`（程序 4.5.5）中的函数 `clusterCoefficient()`，增加一个整数参数  $k$ ，以便它根据当前存在的所有边数，以及顶点之间距离为  $k$  的顶点集合中可能的总边数，计算图的局部聚集系数。当  $k$  等于 1 时，函数产生的结果与函数的无参数版本相同。
- 4.5.20 证明  $k$  阶环图中的聚集系数为  $(2k-2)/(2k-1)$ 。推导出一个  $V$  和  $k$  的函数，计算  $V$  个顶点上的  $k$  阶环图中的平均路径长度。



- 4.5.21 证明  $V$  个顶点的二阶环图中的直径是  $\sim V/4$ 。并证明如果添加连接两个对角顶点的一条边，则直径将减小到  $\sim V/8$ 。
- 4.5.22 进行计算实验，以验证  $V$  个顶点的轮环图中的平均路径长度为  $\sim 1/4 V$ 。将一个随机边添加到轮环图中，并验证平均路径长度减少到  $\sim 3/16 V$ 。
- 4.5.23 为 SmallWorld (程序 4.5.5) 添加函数 isSmallWorld(), 该函数以图作为参数，如果图符合小世界现象的规则 (依据正文定义的具体阈值)，则返回 true，否则返回 false。
- 4.5.24 为 SmallWorld (程序 4.5.5) 实现一个测试客户程序 main(), 它产生正文中给定的输出。程序应该将图文件的名称和分隔符作为命令行参数；打印图的顶点数量、平均度数、平均路径长度和聚集系数；并指出这些数值对于小世界现象是否过大或过小。
- 707 4.5.25 编写一个程序，生成随机连接图和有随机捷径的二阶环图。使用 SmallWorld，从两个模型 (每个具有 1000 个顶点) 生成 500 个随机图并计算它们的平均度、平均路径长度和聚集系数。将你的结果与 4.5 节“具有 1000 个顶点的不同种类的图的小世界参数”表格中的相应值进行比较。
- 4.5.26 编写一个 SmallWorld 和 Graph 的客户程序，生成  $k$  阶环图并测试它们是否符合小世界现象 (首先做练习 4.5.23)。
- 4.5.27 在一个网格图 (grid graph) 中，顶点排列在一个  $n \times n$  的网格中，每个顶点与网格中上下左右的邻接顶点用边相连。撰写一个 SmallWorld 和 Graph 客户程序，来生成网格图并测试它们是否符合小世界现象的规则 (首先做练习 4.5.23)。
- 4.5.28 扩展以上两个练习的解决方案，读入一个命令行参数  $m$ ，并将  $m$  个随机边添加到图中。用你的程序对大约 1000 个顶点的图进行实验，以找到边相对较少的小世界图。
- 4.5.29 编写一个 Graph 和 PathFinder 的客户程序，它将电影 - 演员列表文件的名称和分隔符作为参数，并写入一个新的电影 - 演员列表文件，但删除了未连接到凯文·贝肯的所有电影。
- 708



三阶环图



6×6 网格图

## 创新练习

- 4.5.30 大贝肯数。找到 movies.txt 中拥有最大 (但有限) 凯文·贝肯数的演员 (不与凯文·贝肯连接的演员会是正无穷——译者注)。
- 4.5.31 直方图。编写一个 BaconHistogram 程序，打印凯文·贝肯数的直方图，指出 movies.txt 中有多少个演员的贝肯数为 0、1、2、3……为那些拥有无限大凯文·贝肯数 (不与凯文·贝肯连接) 的人添加一个类别。
- 4.5.32 演员 - 演员图。正如正文中所提到的，另一种计算凯文·贝肯数的方法是建立一个图，其中每个演员 (而不是每个电影) 都有一个顶点，如果两个演员一起出现在电影中，那么它们是相邻的 (参见程序 4.5.6)。通过在演员 - 演员图上运行广度优先搜索计算凯文·贝肯数，与 movies.txt 的运行时间进行比较，解释为什么这种方法慢很多。同时解释一下要怎么在路径中包含电影 (我们的实现就是这么做的)。
- 4.5.33 连通分量。图中的连通分量是指相互连接的最大的一组顶点。编写一个 Graph 客户程序 CCFinder，计算图中的连通组件，类中需要一个将 Graph 作为参数的构造函数，并使用广度优先搜索计算所有连通分量。还需要一个方法 areConnected(v,w)，如果  $v$  和  $w$  在相同的连通分量中则返回 true，否则返回 false。还要添加一个方法 components()，它返回连通分量的数

量（连通分量未必只有一个，因为连通分量强调的是连接尽可能多的顶点——译者注）。

4.5.34 洪水填充 / 图像处理。Picture 是一个 Colour 值的二维数组（见 3.1 节）。一个 blob 是相同颜色的相邻像素的集合。编写一个 Graph 客户程序，其构造函数从给定的图像创建一个网格图（见练习 4.5.27），并支持洪水填充操作。给定像素坐标 col 和 row 以及颜色 color，将该像素的颜色和同一个 blob 中的所有像素的颜色更改为 color。

709

4.5.35 词梯。词梯是指若两个单词仅有一个字母不同，则两个单词相连，由相连单词构成的单词链称为词梯。编写一个程序 WordLadder，从标准输入中读取一个 5 字母的单词列表，将两个 5 字母字符串作为命令行参数，为这两个参数输出最短词梯（若存在的话）。作为一个例子，下面的词梯连接了 green 和 brown：

green greet great groat groan grown brown

编写一个过滤器，从标准输入中获取系统词典的 5 字母单词，或者从本书网站下载列表。（这个游戏最初被称为 doublet，是由 Lewis Carroll 发明的。）

4.5.36 所有路径。编写一个 Graph 的客户程序 AllPaths，其构造函数将 Graph 作为参数，并计算和打印出图中两个给定顶点  $s$  和  $t$  之间所有的简单路径。简单路径（simple path）是指不存在任何重复顶点的路径。在二维网格中，这样的路径被称为自避行走（self-avoiding walks，见 1.4 节）。枚举路径是统计物理学和理论化学中的一个基本问题——例如，模拟线性聚合物分子在溶液中的空间排列。注意：路径的数量可能会是指数级的。

4.5.37 渗透阈值。为渗透开发一个图模型，编写一个与 Percolation（程序 2.4.5）执行相同计算的 Graph 客户程序。估算三角形、正方形和六角形网格的渗透阈值。

4.5.38 地铁图。在东京地铁系统中，路线由字母标记，站点由数字标记，如 G-8 或 A-3。其中仅有某些站点允许换乘。在网络上查找东京地铁地图，开发一个简单的文件格式，然后编写一个 Graph 的客户程序，它读入文件并可以回答东京地铁系统的最短路径查询的问题。如果你愿意的话，做一个巴黎的地铁系统，路线是由一连串名字构成的，当两个站点有同样的名字时，表示可以换乘。

710

4.5.39 好莱坞宇宙的中心。我们可以通过计算每个演员的好莱坞数（Hollywood number）或平均路径长度来衡量把凯文·贝肯作为中心是否合适。凯文·贝肯的好莱坞数是所有演员（在其连通分量中）的贝肯数的平均值。其他演员的好莱坞数也是用相同的方法计算出来的，只需要将这个演员取代凯文·贝肯作为源。计算凯文·贝肯的好莱坞数，找到一个好莱坞数比凯文·贝肯更好的演员。在与凯文·贝肯位于同一个连通分量中，找到具有最好和最差好莱坞数的演员。

4.5.40 直径。顶点的偏心率（eccentricity）是其与任何其他顶点之间的最大距离。图的直径是任何两个顶点之间的最大距离（任何顶点的最大偏心率）。写一个 Graph 的客户程序 Diameter，可以计算顶点的偏心率和图的直径。使用它来查找与 movies.txt 关联的演员 - 演员图的直径。

4.5.41 有向图。实现一个 Digraph 数据类型来表示有向图，其中边的方向是有意义的：addEdge(v,w) 表示从  $v$  到  $w$  添加边，而不是从  $w$  到  $v$ 。用以下两个方法替换 adjacentTo()：一个返回由参数顶点出发的有向边的到达顶点列表，另一个返回到达参数顶点的有向边的出发顶点列表。解释如何修改 Pathfinder 以找到有向图中的最短路径。

4.5.42 随机游走。修改前一个练习中的 Digraph 类，以创建允许平行边的 MultiDigraph 类。对于测试客户程序，运行与 RandomSurfer（程序 1.6.2）匹配的随机游走模拟。

4.5.43 传递闭包。编写一个 Digraph 客户程序 TransitiveClosure，其构造函数将 Digraph 作为参数，如果存在从  $v$  到  $w$  的某个有向路径，其方法 isReachable(v, w) 返回 true，否则返回 false。提

711

示：从每个顶点运行广度优先搜索。

4.5.44 统计抽样。使用统计抽样来估计图的平均路径长度和聚集系数。例如，为了估计聚集系数，挑选 trials 份随机顶点并计算这些顶点的聚集系数的平均值。你的函数的运行时间应该比 SmallWorld 的相应函数快几个数量级。

4.5.45 覆盖时间。一个无向连通图中的随机游走 (random walk) 是指从一个顶点移动到它的邻接顶点之一，在这里每个可能性都有相等的被选择概率 (这个过程是对无向图的随机游走模拟)。编写程序来运行实验，以验证关于访问图中每个顶点所需步数的假设。具有  $V$  个顶点的完全图的覆盖时间是多少？一个轮环图呢？你能找到一系列图，其覆盖时间与  $V^3$  或  $2^V$  成正比吗？

4.5.46 Erdős-Renyi 随机图模型。在经典的 Erdős-Renyi 随机图模型中，我们在  $V$  个顶点上建立一个随机图，包含每个可能的边的概率为  $p$ ，是否包含一个边与其他边无关。构建一个 Graph 客户程序来验证以下属性：

- 连接性阈值 (Connectivity thresholds)：如果  $p < 1/V$  并且  $V$  很大，那么大多数连通分量都很小，最大为对数大小。如果  $p > 1/V$ ，那么肯定会有一个包含几乎所有顶点的巨大连通分量。如果  $p < (\ln V)/V$ ，则图以高概率断开；如果  $p > \ln V/V$ ，则图以高概率连接。
- 度的分布 (Distribution of degrees)：度的分布遵循以平均值为中心的二项分布，所以大多数顶点具有相似的度数。顶点与其他  $k$  个顶点相邻的概率以  $k$  的指数级减小。
- 没有枢纽 (No hubs)：当  $p$  是一个常数时，顶点的最大度数最多是  $V$  的对数。
- 没有局部聚集 (No local clustering)：如果图是稀疏且连通的，则聚集系数接近于 0。随机图不是小世界图。

712

- 短路径长度 (Short path lengths)：如果  $p > \ln V/V$ ，那么图的直径 (见练习 4.5.40) 是对数级的。

4.5.47 网络链接的能量法则。网页的入度和出度遵循用首选连接过程建模的能量法则。假设每个网页只有一个输出链接，所有网页都是依次创建的，一次只创建一个页面，最初的页面只有一个指向自己的链接。以概率  $p < 1$ ，它随机均匀选择现有页面之一并与之建立链接。以概率  $1-p$ ，它链接到一个现有的页面，其概率与该页面的传入链接的数量成正比。这个规则反映了新网页指向流行网页的普遍趋势。撰写一个程序来模拟这个过程，并画出传入链接数量的直方图。

部分解答。入度为  $k$  的网页数与  $k^{-1/(1-p)}$  成正比。

4.5.48 全局聚集系数。为 SmallWorld 添加一个计算图的全局聚集系数的函数。全局聚集系数是与一个公共顶点相邻的两个随机顶点彼此相邻的条件概率。找出局部聚集系数和全局聚集系数不同的图。

4.5.49 Watts-Strogatz 图模型。(见练习 4.5.27 和练习 4.5.28) Watts 和 Strogatz 提出了一个混合模型，这个模型中的顶点与相邻的顶点都存在链接 (人们往往认识地理位置比较近的人)，还有一些随机的远链接。对  $n=100$  阶网格图，绘制当为其添加随机边时，平均路径长度和聚集系数的变化效果。对于  $V$  个顶点的  $k$  阶环图，同样绘制添加随机边的效果图，其中  $V=10\,000$ ， $k$  的上限为  $10 \log V$ 。

4.5.50 Bollobás-Chung 图模型。Bollobás 和 Chung 提出了一种混合模型，它在  $V$  个顶点 ( $V$  是偶数) 上加上一个二阶环图，再加上一个随机匹配 (random matching)。匹配是指每个顶点的度数均为 1 的图。为了生成随机匹配，将  $V$  个顶点随机排序，并在排定的顺序中在顶点  $i$  和顶点  $i+1$  之间添加边。确定此模型中图的每个顶点的度数。使用 SmallWorld，估计根据这个模型产生的  $V=1\,000$  的图的平均路径长度和局部聚集系数。

713  
714

# 计算理论

实体计算机看起来非常复杂，因此难以对它进行分析，但是最终我们建立并且使用了它们，这表明我们实际上是可以了解它们的本质特征的。在本章中，我们通过对一台计算机功能和限制进行严格的研究，以揭示所有已知类型的计算机的共同特征。通过对这些分析，让我们有能力思考下面这些基本问题：

- 某些计算机本质上就比其他计算机更强大吗？
- 我们能用计算机来解决什么类型的问题？
- 计算机能做的事情有极限吗？
- 在有限的资源下，计算机能做的事情的极限是什么？

这些都是很深刻的问题，20 世纪的大半部分时间数学家们都在努力解决这些问题。为了回答这些问题，我们会构建简化和理想化的抽象计算机，同时这些抽象计算机还保留了真正的现代计算机的基本属性。你可能会惊讶地发现，使用这些抽象计算机来进行细致的推理，这些问题是可以找到答案的。

如果“理论”（theory）这个词让你感到恐惧，请不要绝望。我们会使用简单易懂的数学模型。我们会研究一些计算方面的美妙的定理并且让你相信这些定理的真实性。我们不会要求你学习如何证明数学定理（这种能力就像编程，是一种后天学习的技能），但是你需要花时间理解我们展示给你的逐步推理过程。这样做的一个好处是它有可能帮助你成为一个更好的程序员（这个过程有点像调试），但是我们的主要目标是让你学习如何体会一些关于计算的核心事实。

715

我们为什么要深入地考虑这些理论问题呢？难道不应该把这些问题留给计算理论方面最好的专家吗？你可能会对其他科学学科问同样的问题，而计算机科学对于这些问题的答案与物理、化学或者生物学相同。我们用来理解计算的那些模型都是很基础的模型，我们从这些模型中得出来的结论对我们生活的世界有着很深远的影响。许多人可能会为我们能够解决这些基本问题感到惊讶和兴奋。当你看完本章后，你可能会发现自己很想向朋友或者家人解释为什么计算机世界是现在的这个样子。

一个程序员为什么要关心计算理论？在这么一个青少年都努力地将自己的编程技能转化为无数财富的世界中，人们很快就会有计算机是无所不能的这种感觉。但是令人失望的是，计算理论告诉我们事实绝对不是期望的那样。一个人如果没有理解我们在本章提出的理论问题，那么他就不能有效地使用计算机。这个理论提供了一个框架来帮助我们了解我们可以解决什么问题。除此之外，计算理论一直对实践应用有所启发。我们所使用的许多工具（编程语言就是一种）都是这种理论直接产生的。

一个科学家或者一个工程师为什么要关心理论计算呢？因为现在每一个科学家和每一个工程师都是一个程序员，所以上一段所提到的对于他们同样适用。但是这个理论让人感到非常抽象，与我们生活的世界相去甚远。上了年纪的科学家和工程师们倾向于仅仅将计算机当成一个工具，就像一个计算器一样。但是事实完全相反。在解决了将近一个世纪以来与计算

相关的问题之后，研究人员认识到一个明确的转变：在 20 世纪及以前，科学是建立在理解宇宙的数学 (mathematical) 模型上的；在 21 世纪，我们越来越依赖于对计算 (computational) 模型的使用。由计算理论发现的基本思想对科学进步的影响在当今世界是无可争议的。

716 一个人类学家为什么要关心计算理论呢？除去现在很多人类学家也是程序员这个事实外，这个问题的简单答案可能是这个世界上有很多事情值得学习和理解，本章（还有接下来的两章）的关于计算理论的故事绝对是一个值得投入时间和精力话题。从哲学的角度上讲，我们要理解人类与计算机的关系，而且随着计算机在我们的生活中变得无所不在，追求对这种关系的理解的必要性显得越来越迫切。而对计算理论的理解至少是追求对这种关系理解的一个起点。正如你看到的，理论研究的起源之一对于数学的基本哲学问题的追求；而现在则是对于计算的基本哲学问题的追求。

除了这些普遍的原因之外，还有一个特别的理由让我们仔细地对待本书中的计算理论，即它提供了一个历史视角以便于我们理解计算机如何工作的发展历程。同许多其他的科学一样，历史与理论以巧妙的方式相互交织，并能够相互促进对对方的理解。虽然这不是一本历史书，我们也无法全面且详细地讲述历史细节，但每个对计算机感兴趣的人都会因为了解这段历史的亮点而受益。

本章的中心人物是艾伦·图灵。这位英国数学家在 20 世纪 30 年代这段时间在剑桥大学和普林斯顿大学工作。在战争期间的布莱切利园 (Bletchley Park)，许多人认为他通过破译德国的英格玛密码为加快第二次世界大战结束的进程发挥了核心作用。在战后，图灵在计算机科学领域做出了许多有创造力的贡献，其中就包括了对人工智能概念的初步思考。在 50 年代初期，这项工作因他被起诉为同性恋者而停止。随后，图灵因为吃了一个含有氰化物的苹果而死亡（关于他死亡的确切情况是未知且有争议的）。直到 2013 年，英国女王签署了一个死后赦免的法案才推翻了图灵的定罪。在当代，图灵被认为是“计算机科学之父”。你可以通过阅读“问答环节”的参考文献来了解更多图灵的个人事迹。



艾伦·图灵 (1912—1944)

本章我们将重点介绍图灵在 1937 年的伦敦数学学会会议上发表的一篇名为“可计算数及其在决策问题上的应用”的文章所做出的重大贡献。这篇论文被誉为 20 世纪最重要的科学论文之一，本书中 5.2、5.3 和 5.4 节都利用了这篇论文的结论。

为了了解图灵的工作，我们需要转变自己的观点。我们从一个数学家的观点出发，即我们试图将无关的细节排除出去，而只专注于了解在一般环境中应用的核心问题。5.1 节致力于理解图灵的论文奠定基础。

717 在 5.5 节中，我们介绍在计算理论领域中现代研究的焦点，这些焦点集中在图灵逝世之后提出而且至今未解决的一个基本问题上。

## 5.1 形式语言

为了给接下来介绍的重要理论问题奠定基础，开始之前，我们先定义一些非常简单的抽象概念。接下来，我们要介绍一个广泛使用的软件工具及其相关机制，并附上一些应用程序。

**基本定义** 我们从符号 (symbol) 这个抽象概念开始，这个概念是我们构建后继内容的基础。在数学上，符号可以是能够相互区分的任一标识。在刚开始时，把符号当成一个字符或者数字元素是非常有帮助的。下面我们阐明基本定义。

**定义：**一个符号集（或字母表）表示有限个符号的一个集合。

**定义：**一个字符串表示有限符号集的一个序列。

**定义：**一种形式语言是字符串的一个集合（有可能是无限大的），这些字符串都是由同一个符号集合的元素构成。

你可能会觉得这些定义中的前两个对你来说非常简单和熟悉，因此没有必要给出它们的定义。但是它们定义的这些概念是非常基本的，有一个清晰且明确的定义十分重要。第三个定义对于你来说可能是全新的，所以你应该着重理解它。这是一个简单的定义，而且从现在开始我们可以将术语“字符串的集合”和“形式语言”互换使用。

例如，十进制数字集就是一个你十分熟悉的符号集： $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ 。像 2147483648 这样一个整数是由上面符号集组成的一个字符串，所以我们可以将形式语言正整数（positive integer）定义成由这个符号集构成的不以 0 为开头的字符串的集合。

二进制字符串。接下来我们分析一个二进制字符串的集合（由二进制符号集组成的形式语言），它仅仅涉及两个符号。具体这两个符号是什么并不重要，我们可以使用  $\{0, 1\}$ ，但是在本章中我们使用  $\{a, b\}$  来避免与整数的 0 和 1 产生混淆。确定一种形式语言的最简单的方式就是枚举它的字符串。这是一种可以精确确定形式语言的方法，但我们通常还会用非形式的描述来描述语言。例如，我们可以用“长度为 3 的二进制字符串”这样的非形式描述来标识形式语言（字符串的集合）： $\{aaa, aab, aba, abb, baa, bab, bba, bbb\}$ 。

718

|            | 在语言中                                             | 不在语言中                                |
|------------|--------------------------------------------------|--------------------------------------|
| 倒数第二个符号为a  | aa<br>bbbab<br>bbbbbbbbababab                    | a<br>aaaba<br>bbbbbbbbbbbbbb         |
| a与b的数目相等   | ba<br>bbaaba<br>aaaabbbbbbaaaba                  | a<br>bbbaa<br>ababababababab         |
| 回文         | a<br>aba<br>abaabaabaaba                         | ab<br>bbba<br>ababababababab         |
| 字母串中含有abba | abba<br>abaababbabbababbba<br>bbbbbbbbbbabbabbbb | abb<br>bbabaab<br>aaaaaaaaaaaaaaaa   |
| b的数量可以被3整除 | bbb<br>baaaaabaaaab<br>bbabbbaabaaabababaaa      | bb<br>abababab<br>aaaaaaaaaaaaaaaaab |

由二进制符号集组成的形式语言的例子

我们面临的第一个复杂因素是因为语言可以是一个很大甚至是无限大的集合，所以我们无法总是用列出语言中的每一个字符串的方式来描述它。例如，当我们提到回文（palindromes）语言（回文是指从头读到尾和从尾读到头是一模一样的二进制字符串），或者一个字符串中 a 与 b 的数量相等的语言时，你可以很清楚地知道你指的是什么，尽管这两种语言的集合大小是无限的，我们无法列出它们所有的成员。认真思考一下你就会意识到，你之所以非常确定这些语言所表达的意思，是因为你自己能轻易地判断一个给定的二进制字符串是否属于这些语言。我们选取了一些有代表性并能表示出语言特征的字符串，通过判别这些字符串是否在语言中来让你更好地理解。例如，我们将一种语言命名为 b 的数量可以被 3 整除，你可能会产生 a 这个符号是否可以包含在字符串中这种疑问。但是当我们举例说明



719

bbb 和 baaaaabaaaab 属于这种语言而 bb 和 abababab 不属于时，你会对我们的意思有一个更好的理解（a 这个符号是可以忽略的）。这些例子以及其他的一些例子都展示在上表中。

其他符号集。通过对二进制字符串的适当解释，我们可以很轻松地定义与计算问题相关的形式语言。如你所知，我们采用二进制来编码计算机程序，而现在我们也可以对形式语言进行同样的操作。例如，我们可以定义一种名为素数的语言，它是所有用二进制来表示的素数的字符串集合。但是采用十进制的表示法来定义这种语言更为自然。我们没有理由限制自己只用二进制，我们可以使用手头上任何适合用来组成形式语言的符号集：像用于处理文本的标准罗马字母，处理数字的十进制数字，用于处理遗传数据的符号集 {A, T, C, G} 等。当语言的上下文可以使隐晦的符号集能够被清晰地理解时，我们就不用特意指明它。“常用符号集及相关术语”表给出了一些常用的符号集例子。

其他例子。在“更多由不同符号集组成的形式语言的例子”表中我们给出了一些基于不同符号集的形式语言的例子（其中几个会在本节的后面详细描述）。这些例子清楚地表明了形式语言这个概念的广泛适用性。这些形式语言中有一些是基于数学的。例如，一个由十进制表示的字符串集合，这个集合满足对于一个整数  $z$ ，存在正整数  $x, y$ ，当  $n > 2$  时，使得  $x^n + y^n = z^n$ 。这种语言现在被认为是一个空集，但是你可能会难以说服自己这是对的。这是著名的费马大定理的一种表现形式，是一个 300 多年来都未能得到证实的猜想，直到安德鲁·威尔斯（Andrew Wiles）在 20 世纪 90 年代提出了一个证明方法。其他形式语言也有基于人类语言的、Java 的或者基因组学的。我们可以定义一种叫作莎士比亚的语言用于表示莎士比亚的所有戏剧，或者定义一种叫泰勒的语言用于表示泰勒的所有作品。

|         | 符号                                                       | 符号名   | 字符串名 |
|---------|----------------------------------------------------------|-------|------|
| 二进制     | 01（或者ab）                                                 | 位     | 位字符串 |
| 罗马字母    | abcdefghijklmnopqrstuvwxyz<br>ABCDEFGHIJKLMNOPQRSTUVWXYZ | 字母    | 单词   |
| 十进制数    | 0123456789                                               | 数字    | 整数   |
| 特殊符号    | ~`!@#\$%^&*()_-=+{[]}\ :;'"<,>./                         |       |      |
| 键盘输入    | 罗马字母+十进制数+特殊符号                                           | 按键    | 打印稿  |
| 基因编码    | ATCG                                                     | 核苷酸碱基 | DNA  |
| 蛋白质编码   | ACDEFGHIKLMNPQRSTVWY                                     | 氨基酸   | 蛋白质  |
| ASCII   | 见6.1节                                                    | 字节    | 字符串  |
| Unicode | 见6.1节                                                    | 字符    | 字符串  |
|         | 常用符号集及相关术语                                               |       |      |

720

|    | 在语言中                                              | 不在语言中                                             |
|----|---------------------------------------------------|---------------------------------------------------|
| 回文 | madamimadam<br>amanaplanacanalpanama<br>amoraroma | madamimbob<br>madam, i'm adam<br>not a palindrome |
| 奇数 | 3<br>101<br>583805233                             | 2<br>100<br>2147483648                            |
| 素数 | 3<br>101<br>583805233                             | 0003<br>100<br>2147483648                         |

更多由不同符号集组成的形式语言的例子



|                                                              |                                                                       |                                      |
|--------------------------------------------------------------|-----------------------------------------------------------------------|--------------------------------------|
| 对于整数 $z$ , 能找到整数 $x$ 、 $y$ 满足 $x^2 + y^2 = z^2$              | 5<br>13<br>9833                                                       | 2<br>16<br>9999                      |
| 对于整数 $z$ , 能找到整数 $x$ 、 $y$ , 当 $n > 2$ 时满足 $x^n + y^n = z^n$ | 没有整数可满足                                                               | 所有整数                                 |
| 氨基酸编码                                                        | AAA ACA ATA AGA CAA CCA<br>CTA CGA TCA TTA GAA GCA GTA<br>GGA AAC ACC | CCC<br>AAAAAAAAA<br>ABCDE            |
| 美国电话号码                                                       | (609) 258-3000<br>(800) 555-1212                                      | (99) 12-12-12<br>2147483648          |
| 英语单词                                                         | and<br>middle<br>computability                                        | abc<br>niether<br>misunderestimate   |
| 语法正确的英文句子                                                    | This is a sentence.<br>I think I can.                                 | xya<br>a b.c.e??<br>Cogito ergo sum. |
| 合法的Java标识符                                                   | a<br>class<br>\$xyz3_XYZ                                              | 12<br>123a<br>a((BC))*               |
| 合法的Java代码                                                    | public class Hi {<br>public static void<br>main(String[] args) { } }  | int main(void)<br>{ return 0; }      |

更多由不同符号集组成的形式语言的例子 (续)

721

规范问题。我们可以用非形式化的语言描述来处理某些情况 (如回文和素数), 但还是有一些不足以处理的情况 (如英语句子和 Java 程序)。为什么不使用精确、完整的定义呢? 这是问题的关键所在! 我们的目标是使用精确且完整的形式语言。这个目标被称为形式语言的规范问题。

那么我们如何精确且完整地定义一种形式语言呢? 我们的非形式描述能够解决一些问题, 但是要知道, 总有一些是不能通过语言精确描述的。我们可以清楚地表达一些形式语言, 但这不是全部。例如, 我们能否规定素数前面可以有前导零, 就比如字符串 000017 是否可以作为素数这种形式语言的一个元素? 请注意, 这个决定有可能会对每个素数都对应无数个字符串。这样的细节使得定义一种形式语言成为一件具有挑战性的事情。但是这个挑战的关键并不在于这些细节, 而是规范问题。事实证明, 我们可以找出某些特定类别的形式语言, 它们非常易于给出严格的规范。稍后我们会针对其中一种基础的类别——通常简称为正则语言 (regular language), 解决其规范问题。

识别问题。一旦我们有一种方法指定形式语言的规范, 我们就会遇到下面这种问题: 给定一种语言  $L$  和一个字符串  $x$ , 回答  $x$  是否在  $L$  里面? 这种情况被称为形式语言的识别问题。要解决识别问题, 你需要拥有一台计算机 (除了计算机还有谁可以告诉你一个十亿位级的字符串是否属于回文或者是否拥有相等数量的  $a$  和  $b$ ?)。除此之外, 你需要有一定的数学知识, 一定的自然语言语法知识, 以及数不清的其他领域的知识。幸运的是, 我们同样可以找出某些特定类别的形式语言, 可以很容易地解决其识别问题, 同时又非常有用。

722

我们将通过学习一类重要的语言 (正则语言) 来开始我们对形式语言的学习。我们可以为正则语言找到规范和识别的简单的解决方案。然后我们可以看到这些解决方案是非常重要的工具, 这些工具被广泛用于各种实际应用中, 这其中涉及自然语言 (比如英语)、编程

语言（比如 Java）、基因编码以及许多领域。在这之后的 5.2 节，我们会回到对基本理论问题的学习。值得注意的是，我们用于解决正则语言的规范问题和识别问题的解决方法，经过一些相对简单的扩展后，不仅可以大大扩展我们可以处理的形式语言，而且将我们直接引导到了计算问题基本原理的核心部分。事实上，形式语言与计算之间有着密切的联系。

我们将要介绍一些用于描述形式化语言的机制，这些机制是简单、紧凑且优雅的。它们遵循下面这两种方法之一。第一种方法是基于语言的：在我们将要描述的语言本身的字符之外，我们再分配一些字符用来描述语言的规范。第二种方法是基于机器的：我们描述了一类抽象机器，每一个机器都对应于一种语言的规范和识别问题。

**正则语言** 为了方便理解形式语言，我们将研究一类被称为正则语言的简单形式语言。为了解决正则语言的规范和识别问题，我们开发了一种基于语言的方法和两种不同的基于机器的方法。然后，在本节的最后，我们会讨论这三者的关系。

**基本操作。**由于形式语言是一个字符串的集合，因此我们可以使用对集合的基本操作来有效描述形式语言的规范。特别是我们使用了并集、连接和闭包这些运算符，我们首先对这些操作进行简单介绍。

当且仅当一个字符串属于两个字符串集合的一个或者两个，这个字符串就属于它们的并集。我们使用符号  $R \mid S$  来表示两种形式语言  $R$  和  $S$  的并集。例如：

$$\{a, ba\} \mid \{ab, ba, b\} = \{a, ab, ba, b\}$$

我们可以以任何顺序列出一个语言的字符串成员，它们都表示同一个集合，但是在集合中不会包含重复的字符串成员。

[723]

两个字符串的连接是将第二个字符串附加到第一个字符串上生成的字符串。例如  $abb$  和  $aab$  的连接就是  $abbaab$ 。更一般的，由连接两种形式语言  $R$  和  $S$  的  $RS$  是将  $R$  中的任意一个字符串与  $S$  中的任意一个字符串连接所生成的字符串的集合。例如：

$$\{a, ab\} \{a, ba, bab\} = \{aa, aba, abab, abba, abbab\}$$

同样的，生成的结果集合中不包含重复的成员（在上面例子中， $a$  连接  $ba$  和  $ab$  连接  $a$  都可以生成  $aba$ ）。

闭包指从一种语言中选取 0 个或多个字符串，用这些字符做连接操作所得到的所有字符串组成的集合。如果  $R$  是一个非空集合，那么符号  $R^*$  就可以表示如下的一组无限多个的字符串：

$$R^* = \epsilon \mid R \mid RR \mid RRR \mid RRRR \mid RRRRR \mid RRRRRR \dots$$

注意，每当我们从  $R$  中提取一个字符串时，我们可以使用  $R$  中的任意一个字符串。例如， $(a|b)^*$  指定了所有二进制字符串的集合。这里， $\epsilon$  指的是空字符串——由 0 个字符组成的字符串。

如果我们有一个含有多个运算符的正则表达式，那我们的运算符应该按一种什么样的顺序运算呢？与算术表达式一样，我们通过括号或运算符的优先级顺序来解决这种模糊问题。对于正则表达式，闭包的优先级比连接高，连接的优先级比并集高。例如，正则表达式  $b|ab$  指定的集合是  $\{b, ab\}$  而不是集合  $\{bb, ab\}$ 。类似的，正则表达式  $a|b^*$  指定的是与  $a|(b^*)$  相同的集合，而与  $(a|b)^*$  不是同一个语言。

加上括号的正则表达式

GCG(CGG|AGG)\*CTG

并集

闭包

正则表达式的分解

正则表达式。正则表达式是一个由符号组成的字符串，它可以用来规范一种形式语言。我们使用并集、连接还有闭包这些用在集合上的运算符，以及用于表达操作优先级的括号来递归地定义什么是正则表达式（以及它们的含义）。具体来说，每一个正则表达式要么是符号表中的某一个符号（用于表示包含这个符号的单元元素集合），要么就是由下列运算符得到的组合（R 和 S 都是正则表达式）：

- 并集：规定  $R|S$  为集合 R 和集合 S 的并集。
- 连接：规定  $RS$  为集合 R 和集合 S 的连接。
- 闭包：规定  $R^*$  为集合 R 的闭包。
- 括号：规定  $(R)$  与集合 R 相同。

724

这些定义都包含了一个隐式的假设，即假定语言的符号集之中是不包含符号 “|”、“\*”、“(” 和 “)” 的。我们将这些符号称为元符号（metasymbol），并在稍后讨论如何处理包含这些符号的语言。

正则语言。以上递归定义不仅可以用于建立任意复杂度的正则表达式，还可以准确地定义这些正则表达式的含义。每个正则表达式都能够完整地描述一些形式语言，但不是每一种形式语言都可以被正则表达式规范地描述。稍后，我们会正式地证明这个事实。这种证明的过程是通过找到一种不能被任何正则表达式所规范的形式语言来实现的。但是可以用一些正则表达式来规范的这类形式语言是非常重要的，对它有如下定义：

**定义：**当且仅当一种语言能被一条正则表达式描述时，它是正则的。

下面表格中给出了一些正则语言的例子，并且还附有描述这些正则语言的正则表达式，以及属于这些正则语言的字符串实例和不属于这些正则语言的字符串实例。那么我们如何确保每一个正则表达式规范的正则语言与非正式描述的相同呢？一般来说，对每一个正则表达式我们都需要进行相应的证明，而且做这样的证明往往有一定的难度。正则表达式的重要意义就在于它们可以让我们摆脱非形式的描述，因为它们提供了一种用来规范语言的方式，这种方式对于那些重要的应用来说十分自然并且严谨。无论我们对非形式规范的准确性是否有信心，每个正则表达式都是对一些正则语言精确且完整的描述。事实上，我们可以写下一个形式正确的正则表达式却不需要去想它究竟描述了哪一种语言，如  $(ab^*|aba)^*(ab^*a|b(a|b))^*$ 。

为了熟悉正则表达式，你应该花时间让你自己相信表中的每个正则表达式确实正确描述了它所声明的语言。首先你不仅仅需要确定通过验证的字符串符合非形式描述并且被正则表达式所描述，还要确定不属于这种语言的那些字符串不符合非形式描述并且没有被正则表达式所描述。你的目标是理解正则表达式不仅规范了语言中的所有字符串，并且仅有这些字符串能被这条正则表达式描述。下面的段落给出了关于某些语言的细节。

725

| 正则语言      | 正则表达式                          | 在语言中                                      | 不在语言中                           |
|-----------|--------------------------------|-------------------------------------------|---------------------------------|
| 二进制符号集    |                                |                                           |                                 |
| 倒数第五个符号是a | $(a b)^*a(a b)(a b)(a b)(a b)$ | aaaaa<br>bbbabbbb<br>bbbbbbababababa      | a<br>bbbbbbba<br>aaaaaaaaaaba   |
| 包含字符串abba | $(a b)^*abba(a b)^*$           | abba<br>aababbabababbba<br>bbbbbbbababbbb | abb<br>bbabaab<br>aaaaaaaaaaaaa |

基于不同符号集的正则表达式的例子

|                 |                                           |                                      |                                      |
|-----------------|-------------------------------------------|--------------------------------------|--------------------------------------|
| 不包含字符串bbb       | $(bba ba a^*)^*(a^* b bb)$                | aa<br>ababababbaba<br>aaaaaaaaaaaaab | bbb<br>ababbbbabab<br>bbbbbbbbbbbbbb |
| 符号b的数量<br>是3的倍数 | $a^* (a^*ba^*ba^*ba^*)^*$                 | bbb<br>aaa<br>bbbaababbaa            | b<br>baaaaaaab<br>baabbbaaaaab       |
| 十进制数字           |                                           |                                      |                                      |
| 能被5整除的正整数       | $5 (1 2 \dots 9)(0 1 \dots 9)^*(0 5)$     | 5<br>200<br>9836786785               | 1<br>0005<br>3452345234              |
| 正三元数            | $(1 2)(0 1 2)^*$                          | 11<br>2210221                        | 011<br>19<br>9836786785              |
| 小写字母            |                                           |                                      |                                      |
| 包含字符串spb        | $(a b c  \dots  z)^*spb(a b c  \dots  z)$ | raspberry<br>crispbread              | subspace<br>subspecies               |
| 只用键盘的<br>最上面一行  | $(q w e r t y u i o p)^*$                 | typewriter<br>reporter               | alfalfa<br>paratrooper               |
| 基因编码            |                                           |                                      |                                      |
| 脆性X染色体<br>综合征图谱 | $GCG(CGG AGG)^*CTG$                       | GCGCTG<br>GCGCGGCTG<br>GCGCGGAGGCTG  | GCGCGG<br>CGGCGGCGGCTG<br>GCGCAGGCTG |

726

基于不同符号集的正则表达式的例子 (续)

当你玩填词游戏或者玩单词游戏的时候，你可能会遇到下面这样的问题：“一个 8 个字母的单词，它的中间两个字母是 hh，这个单词是什么？”一旦你了解本节中使用正则表达式的描述规则，你将从一个完全不同的角度去解决填词游戏和单词游戏的问题。

在基因组学中，基于符号集 {A, T, C, G} 组成的正则表达式用来描述基因的性质。例如，人类基因组中有一个可以用正则表达式  $GCG(CGG|AGG)^*CTG$  来描述的区域，这个区域中  $CGG/AGG$  模式在不同的个体之间高度不一致。已知有一种会引起智力低下及其他症状的遗传性疾病与  $CGG/AGG$  基因的大量重复有关。正则表达式在实践中被广泛地用于解决这类重大的科学问题。

在信息处理中，我们始终关注于以一种精确且完整的方式描述信息。例如，当你将名称、地址和其他信息输入网上的表格时，处理信息的程序的第一个操作是检查你输入的内容是否有意义。如果你在一个期望你输入数字的地方输入字母，或者在一个期望你输入电话号码的地方输入符号 \$，处理信息的程序都会标记为错误并要求你改正它们。我们很快就会看到如何使用正则表达式来描述熟悉的低层次的抽象概念，如日期、信用卡号以及 Java 标识符。这样的描述可以方便我们从数据库中将相关的数据提取出来（如科学实验的结果）。

在计算机科学中，正则表达式无处不在。正如你所看到的，它们是很多应用程序都有的功能——Java 本身就具有许多基于正则表达式的功能。正则表达式也是将以高级程序语言（如 Java）编写的程序编译成机器语言过程的第一步。更重要的是，正如我们强调的那样，正则表达式是在解决关于计算的基本问题的道路上的第一步。

一般来说，我们都知道可以用一条正则表达式来规范的任何一种语言都是正则语言——

这是我们的定义。但是以某种其他方式规范的语言呢？所有回文组成的一个集合是正则语言吗？本节的目标之一是提高你对语言分类的理解。虽然有很多有趣且有用的正则语言的例子，但还有很多有趣且有用的语言不属于正则语言。后面，我们会介绍一个比正则表达式更强大的规范系统来解决非正则语言的问题。

727

正则表达式的识别问题。正如我们所说，另一个根本问题是：给定一个二进制字符串，我们要如何才能知道这个字符串是否属于给定的正则表达式所规范的语言呢？例如，字符串 `abaaabbbbbababbbba` 是否属于正则表达式  $(ab^*|bab)^*(bb^*b|(b(a|b)))^*$  规范的语言呢？这是一个正则语言识别问题的例子。正则表达式的一大特点在于它能够帮助我们解决一大类语言（正则语言）的精确描述问题，但是它们并没有解决这些语言的识别问题。当然，我们需要先解决在正则语言中的识别问题，然后才能解决在更加困难的类别语言之中的这个问题。就比如所有素数组成的集合或者所有 Java 程序组成的集合中的这个问题。

我们将在本节的后面用抽象机器来介绍一个解决正则语言识别问题的方法。事实上，可以使用 Java String 库中的 `matches()` 方法实现一个简单的解决方案，所以我们首先描述怎么实现。如果 `s` 表示任意一个 Java 字符串并且 `re` 表示任意一个正则表达式，那么如果 `s` 属于被 `re` 所规范的语言则 `s.matches(re)` 为真，否则为假。程序 5.1.1 使用这种方法来解决识别问题：它采用正则表达式作为命令行参数，对于每一个符合标准输入的字符串，如果这个字符串属于被正则表达式所规范的语言，则打印 Yes，如果不属于则打印 No。轻松执行此类检查的能力非常有用，所以在回到理论介绍之前，我们会先进行一些扩展和范化，以便你能更好地熟悉正则表达式。同时我们还会向你介绍一些 Java 标准库中不可或缺的编程工具。

除了识别问题，正则表达式提供的完整和精确的规范会立即将我们引导到其他几个自然而明确的问题。例如，给定两个正则表达式，我们如何检查它们是否规范了相同的语言。我们很容易就能够检查出  $a(b|ab|aab)$  和  $(a|aa)(b|ab)$  规范了相同的语言，但是  $((abb|baaa)^*(ba^*|b^*a))^*$  规范的语言和  $(ab^*|bab)^*(bb^*b|a(a|b)))^*$  相同吗？这个问题是正则语言的等价问题（equivalence problem）。你会如何处理那些可以复杂到几千个符号长度的正则表达式？同样的，当我们从字典中查询 8 个字母长度且中间两个字母为 `hh` 的单词时，我们会求两种语言的交集（intersection），即同时属于两种语言的字符串。正则表达式让我们可以精确地定义这样的问题，但是解决这些问题是完完全全另一回事。如果你发现这些问题令你着迷（很多人都这样），你很可能对计算理论的进一步学习乐在其中。这些问题仅仅是冰山一角而已。

728

**广义的正则表达式** 我们对正则表达式的定义是一个包括了用于描述正则语言的四个基本运算符（连接、并集、闭包和分组）的最小的集合。在实践中，我们经常对这个集合进行各种各样的扩展。在这里，我们简要介绍在 Java 中发现的一些正则表达式的扩展，主要分为三大类：

- 对符号集进行扩展
- 求并集操作的缩写符号
- 对闭包运算符的扩展

为了简洁起见，我们将 Java 的正则表达式称为广义 RE 或者 Java RE，这两者之间不需要做更精细的区分。在其他语言和其他应用中也广泛定义了类似的机制，但是“广义”的确切定义是如何构成的并没有一个统一的说法。

程序5.1.1 有效性检查（正则表达式识别）

```

public class Validate
{
 public static void main(String[] args)
 {
 String re = args[0];
 while (!StdIn.isEmpty())
 {
 String s = StdIn.readString();
 if (s.matches(re)) StdOut.println("[Yes]");
 else StdOut.println("[No]");
 }
 }
}

```

这个程序说明了Java中使用正则表达式的机制：Java String库中的matches()方法解决了正则表达式的识别问题。main()函数将正则表达式作为命令行参数，对于标准输入的每个字符串，如果它符合正则表达式描述的语言规范，打印Yes，否则打印No。

```

% java Validate "(a|b)*a(a|b)(a|b)(a|b)(a|b)"
bbbbbbba
[Yes]
bbbbbbba
[Yes]

% java Validate "a*(a*ba*ba*ba)*"
bbbaababbba
[Yes]
baabbbbaaaab
[No]

% java Validate "CGG(CGG|AGG)*CTG"
CGGCGGAGGCTG
[Yes]
CGGCGGCGGCTG
[No]

```

所有的广义概括都具有一个共同特征，就是每个广义的正则表达式都描述了一种正则语言。也就是说，（在原则上）你可以将任何一个广义 RE 转换为一个像我们前面一直在提的标准正则表达式（可能形式上会更为烦琐）。这个限制是具有讽刺意味的，因为广义正则表达式实际上并没有将正则语言广义化，而仅仅是一种用来描述正则语言的语言。在适当的时候，我们会介绍真正的广义化。同时，我们需要注意，许多系统（包括 Java）支持的正则表达式扩展不是严格遵守这个限制的。我们稍后将回到这个话题。

**符号集。**符号集的扩展和广义化是最明显的。Java 正则表达式中的符号是没有任何限制的 Unicode 字符。但是当我们完全以这种方式来对符号集进行扩展时，我们会碰到一个固有的问题——我们需要一个转义机制（escape mechanisms）来让我们可以将元符号 |、\*、( 和 ) 与出现在语言字母表中的符号区分开来。具体来说，用 \\ 表示 |、\\\* 表示 \*、\\( 表示 (、\\) 表示 )。在后面描述的其他扩展中，在将所使用的元符号当作语言符号来使用的时候，都需要加 “\\” 转义。

**缩写符号。**符号集中有了大量符号后会立即产生对符号缩写的需求，以满足我们通过一组特殊符号来描述一系列复杂的语言。例如，通配符 (.) 可以用于表示符号集中的任意符

号；它可以是一个符号集中所有符号求并集操作所得的一长串符号的缩写。例如，当处理十进制数字时，比起输入 0|1|2|3|4|5|6|7|8|9，我们当然更愿意直接输入通配符“.”。广义 RE 提供了一些类似这样的缩写，例如：

- 元符号 ^ 表示一行的开头，而 \$ 表示一行的结尾。
- 用方括号 ([]) 包含一个列表或者范围，用于表示这个列表或范围的所有符号。
- 如果方括号中的第一个字符是 ^，则指的是不在这个列表或范围内的字符。
- 由反斜杠后面跟着字符组成的若干个转义字符用于表示一些特殊含义的字符。例如，\s 表示空格符。

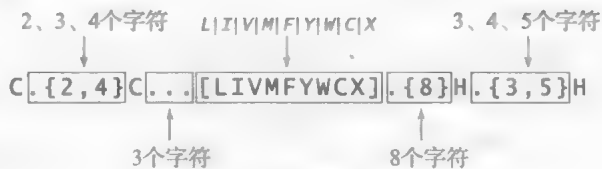
例如，^text\$ 表示单词“text”独占一行，[a-z] 表示所有小写字母，[0-9] 表示十进制数字，[^aeiou] 表示不是小写元音的字母，[A-Z][a-z]\* 表示首字母大写后跟若干个小写字母组成的字符串。

闭包。闭包操作实在是太宽泛了，所以很难在实际应用中直接使用。因此，Java RE 有下面这些选项来对重复的次数进行限制：

- 一次或者更多 (+)
- 零或一次 (?)
- 精确 n 次 ({n})
- 处于 m 和 n 之间 ({m, n})

例如，[^aeiou]{6} 表示不包含小写元音字母的、由 6 个字母组成的单词，如 rhythm 和 syzygy。

这些符号中的每一个都是标准正则表达式的缩写，尽管一个正则表达式可能会非常长。例如 [^aeiou] 这个广义正则表达式是一个对所有非元音字符求并集得到的字符串的缩写，而 [^aeiou]{6} 是该字符串长度为 6 的缩写。从原理上考虑，你可能会认为 Java 处理广义 RE 的方式是首先将它们转换为一个长的标准正则表达式来处理；在实践中，每个扩展都是在对算法设计进行优化的基础上实现的。



一个广义正则表达式的解剖

从这里开始往后你会遇见许多的 Java RE，我们在后面会给出一些示例。像前面一样，你的任务是认真学习这些例子中的每一个，想想为什么给出的例子在描述的语言里，而其他的字符串不在，并尝试了解正则表达式如何完成它们的工作。

**应用** 那么我们应该如何在实际应用中使用正则表达式呢？这些正则表达式出现在许多场景中，其中的一些我们现在就会介绍。从根本上说，这些方案都是通过用相对较短的正则表达式来描述相对较大（甚至无穷）的字符串集合，从而实现简化描述。通过使用正则表达式，一个人或一个程序可以将一整个字符串集合当成一个整体。

**有效性检查。**程序 5.1.1 对于广义 RE 是有效的（并且更为有用）。你可能不知道在你使用网络的时候你经常遇到正则表达式识别问题。当你在商业网站上输入一个日期或者是一个账号的数字时，输入处理程序必须检查你的输入是否是正确的格式。执行此类检查的一种方法是写一段代码来检查所有的情况：如果你要输入钱数（单位为美元），代码可能会检查你输入的首个符号是不是 \$、你的 \$ 符号后面跟的是不是一组数字等。而更好的办法是定义一



个正则表达式以能够描述所有合法输入组成的集合。然后，检查你的输入是否合法这个问题会精确地转化为一个正则表达式识别问题：

你输入的字符串在正则表达式描述的语言里吗？用于常见检查类型的正则表达式库在网络上已经有很多了，因为这种类型的检查已经被广泛地使用了。通常来说，一个正则表达式比检查所有可能情况的程序更能精确且简洁地表达出所有有效字符串的集合。下面的几个例子很好地说明了这一点。

```
% java Validate "\\$[1-9][0-9]*\\. [0-9][0-9]"
$22.99
[Yes]
$1,000,000.00
[No]

% java Validate "ATG(...)+(TAG|TAA|TGA)"
ATGCGCCTGCGTCTGTACTAG
[Yes]
ATGATTGTAG
[No]
```

| Unicode                                | 正则表达式                                           | 在语言中                                                                 |
|----------------------------------------|-------------------------------------------------|----------------------------------------------------------------------|
| Java标识符（部分）                            | <code>[\$_a-zA-Z][\$_a-zA-Z0-9]*</code>         | <code>i</code><br><code>_</code><br><code>\$</code><br>System        |
| 电子邮件地址（部分）                             | <code>[a-zA-Z]+@([a-zA-Z]+\.)+...</code>        | XYZ@yahoo.com<br>rs@princeton.edu<br>xx@whitehouse.gov<br>whitehouse |
| 十进制数字加分隔符                              |                                                 |                                                                      |
| 美国社会安全号码                               | <code>[0-9]{3}-[0-9]{2}-[0-9]{4}</code>         | 330-12-3456<br>213-44-5689                                           |
| 美国电话号码                                 | <code>\([0-9]{3}\) [0-9]{3}-[0-9]{4}</code>     | (800) 555-1212<br>(609) 258-4345                                     |
| 美元数量（部分）                               | <code>\$[1-9][0-9]*\\. [0-9][0-9]</code>        | \$22.99<br>\$1000000.00<br>\$0.01                                    |
| 基因编码                                   |                                                 |                                                                      |
| 潜在的基因                                  | <code>ATG(...)+(TAG TAA TGA)</code>             | ATGATGATGATGTGA<br>ATGAAATAG<br>ATGCGCCTGCGTCTGTACTAG                |
| 氨基酸                                    |                                                 |                                                                      |
| C <sub>2</sub> H <sub>2</sub> 型锌指结构域特征 | <code>C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H</code> | CCCCCCCCCCCCCHHHHH<br>CAASCGGPYACGGWAGYHAGWH                         |

#### 基于不同符号集的广义正则表达式的示例

计算生物学。正如我们已经看到的，研究人员已经开发了许多编码方式，以便我们对基因数据进行操作和分析。我们前面提到的（基于 ATCG 符号集）基因编码方式最简单；另一种方法是使用标准的单字母编码来表示氨基酸，这个编码使用“ACDEFGHIKLMNPQRSTVWY”20 个字母组成的符号集。有关生物学方面的科学理论在本书中不会涉及，但是这里有一个使用这种编码的例子，一个 C<sub>2</sub>H<sub>2</sub> 型锌指结构域特征被定义为：

- 一个 C 后面跟两个、三个或者四个氨基酸；
- 后面紧接：一个 C 后面跟三个氨基酸；
- 后面紧接：L、I、V、M、F、Y、W、C 或者 X 其中一个后面跟八个氨基酸；
- 后面紧接：H 后面跟三个、四个或者五个氨基酸；

- 后面紧接：H。

不难想到，广义正则表达式： $C.\{2,4\}C\dots[LIVMFYWCX].\{8\}H.\{3,5\}H$  能够更为紧凑地表达出这个定义。例如，字符串 CAASCGGPYACGGWAGYHAGWH 是一个  $C_2H_2$  型锌指结构域，因为它以如下方式组成：

- C 后面跟 AAS（在二到四个氨基酸之间）；
- 后面紧接：C 后面跟 GGP（三个氨基酸）；
- 后面紧接：Y（一个属于集合  $\{L, I, V, M, F, Y, W, C, X\}$  的氨基酸），后面跟 ACGGWAGY（八个氨基酸）；
- 后面紧接：H 后面跟 AGW（在三到五个氨基酸之间）；
- 后面紧接：H。

对于这个广义 RE，我们可以使用程序 5.1.1 来测试给定的字符串是否是一个  $C_2H_2$  型锌指结构域：

```
% java Validate "C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H"
CAASCGGPYACGGWAGYHAGWH
[Yes]
CAASCGGPYACGGWAGYHAGWH
[No]
```

**搜索。**在现代计算机系统中，我们看起来总是在使用搜索这个功能。你肯定对在网页上进行搜索、查找文件和使用各种应用程序内置的基于字符串进行搜索的功能感到非常熟悉。你常用的方式很可能是输入一个字符串，然后寻找与这个字符串相匹配的项。有了正则表达式，你可以更灵活且更精确地进行搜索。对于计算机科学家来说，一个名为 `grep` 的搜索工具是一个有用的软件工具。这个工具由肯·汤普森（Ken Thompson）在 20 世纪 70 年代为 UNIX 系统开发，并且在现代的大部分计算机系统中仍然被当作一个常用命令使用。程序 5.1.2 是使用 Java 内置工具实现 `grep` 的一种方案，作为一个过滤器，它以正则表达式作为命令行参数，若标准输入中包含符合正则表达式语言的子字符串，则在标准输出打印这一行。

734

我们可以使用 `Grep`（或者内置的 `grep`）来快速完成各种搜索任务，比如：

- 查找所有包含特定标识符的代码行。
- 查找字典中符合某种模式的所有单词。
- 查找基因组中包含某种模式的片段。
- 查找某个演员出演过的电影。
- 从一个数据文件中提取包含特定字符的行。
- 以及许多其他的任务。

本节的练习当中提供了在各种领域运用搜索的例子，你肯定会对其中的某一些领域感兴趣。即使是习惯于做这些练习的经验丰富的程序员，他们仍然会发现 `grep` 是一个不可缺少的工具。现在许多高级应用程序都内置了搜索功能，使得用户常常忽视了基于正则表达式匹配的搜索。

通过信息来进行验证和搜索都是一些基本操作，采用广义 RE 来进行这些操作比不使用广义 RE 会简单很多。我们使用这些例子是为了说明，理解和使用正则表达式是任何想要有效利用计算的人必须做到的事情。在现代科学，工程和商业应用中都充斥着大量的数据，所以运用模式匹配工具的技能在我们的成长中发挥着重要的作用。正如我们在程序 5.1.1 和程序 5.1.2 中所阐释的，Java 工具让在一个应用中包含正则表达式的搜索变得容易（同样适合

于其他现代编程环境), 所以你可能会发现对正则表达式的应用正在飞速增长。

更重要的是, 正则表达式展现了一个基本范式。这个想法的基本概念是先用一个形式规则来描述一种语言, 然后识别一个给定的字符串是否属于给定的语言。要想充分了解它, 你首先要了解识别过程的工作原理。

为了有效地使用正则表达式, 我们需要解决识别问题。思考一下如果不用 Java 的 `matches()` 方法, 你会如何实现程序 5.1.1? 即使对于二进制符号集上的基本正则表达式也应该如此。那么我们如何编写一个程序, 以一个正则表达式和一个字符串为输入, 并判断该字符串是否在正则表达式定义的语言中呢? 为了解决这个问题, 我们开始走上研究计算机科学中核心基本概念的道路。

程序5.1.2 广义RE模式匹配

```
public class Grep
{
 public static void main(String[] args)
 {
 String re = args[0];
 while (StdIn.hasNextLine())
 {
 String line = StdIn.readLine();
 if (line.matches("." + re + "."))
 System.out.println(line);
 }
 }
}
```

这个程序实现了经典的软件工具grep: 它采用正则表达式作为它的参数, 若标准输入中包含符合正则表达式语言的子字符串, 则打印标准输出行。为了简单起见, 这个实现中使用了Java String库中的`matches()`方法; Java中的`Pattern`和`Matcher`类可以为复杂的正则表达式和巨大的文件提供更有有效的实现方法(见程序7.2.3)文件`words.txt`可以在本书官网上找到, 这个文件包含了字典里的所有单词, 每个一行。在4.5节里有对文件`movies.txt`的描述

```
% java Grep class < Grep.java
public class Grep
% java Grep "[tuv][def][wxy].$" < words.txt
text
% java Grep "A...hh..." < words.txt
withheld
withhold
% java Grep "Bacon, Kevin" < movies.txt | java Grep "Sedgwick, Kyra"
Murder in the First (1995)/ ... /Bacon, Kevin/ ... /Sedgwick, Kyra
Woodsman, The (2004)/ ... /Bacon, Kevin/ ... /Sedgwick, Kyra
```

**抽象机器** 为了解决诸如正则表达式的识别等问题, 我们想出了一个简单的计算模型。起初, 这个模型在你看来与正则表达式完全无关, 但是请你放心, 这是解决这些问题的第一步。

抽象机器是一个形式语言识别问题的计算模型。最普遍的情况下, 抽象机器不过是一个将输入的一个字符



一个抽象机器

串映射到单个输出位的数学函数。我们通常会考虑一个更为详细的模型，将输入符号分开考虑并逐个处理。具体来说，对于任何给定的形式语言，我们设想这么一个设备，这个设备与外部设备的接口有三个组成部分：一个开关，一个可以一次从纸带中读取（也可能是写入）一个符号的输入/输出设备，以及至少两个状态的指示灯（Yes 和 No）。我们将一个给定的字符串放入纸带以备加载，然后打开机器。这时我们希望机器能够读取输入纸带上的内容，并且如果读到的字符串在这个机器的形式语言中，指示灯“Yes”亮起，如果不是的话指示灯“No”亮起。如果指示灯“Yes”亮起，我们说机器“接受”这个字符串，或者说这个字符串匹配，如果指示灯“No”亮起，我们说机器“拒绝”这个字符串或者这个字符串不匹配。因此，一个抽象机器描述了一种形式语言（它所接受的所有字符串的集合）并为该语言的识别问题提供了一个抽象的解决方案（将一个字符串放在纸带上，打开机器并查看“Yes”指示灯是否亮起），这就是我们常说的一台机器可以识别一种语言。我们将思考一些抽象机器，这些机器在操作纸带的能力以及输入/输出设备和指示灯的可用性方面有所不同。

我们现在关注的是抽象机器在计算理论中的使用，但是你应该注意到这个做法是合理的，因为抽象机器适合对所有类型的实际计算问题进行建模。例如，当你使用自动取款机时，你按按钮的顺序队列会对应一个输入字符串，接受指示符对应你的钱已经被发放的情况，拒绝指示符对应你收到一个金额不足的提示。或者我们考虑这么一种情况，输入的字符串是一个十进制整数，并且我们希望机器在当且仅当这个整数为素数时亮起“Yes”指示符（否则就在纸带上写入它的最大因数）。我们在本书开头提到的 Java 程序的总览也可以看作一个抽象机器。这个想法很普遍，它包括了我们可能会遇到的任何类型的计算。我们将在本章中更具体地介绍一些其他例子。

737

一台抽象机器要如何决定是点亮“Yes”指示灯还是“No”指示灯呢？我们希望这台机器必须读取部分或全部输入并进行一些计算。那么应该进行什么样的计算呢？抽象机器的特征是由我们为它定义的基本操作所决定的。我们的目标是剥离那些不必要的细节，并专注于研究那些最简单的机器，这些机器仅使用尽可能简单的一组基本操作。

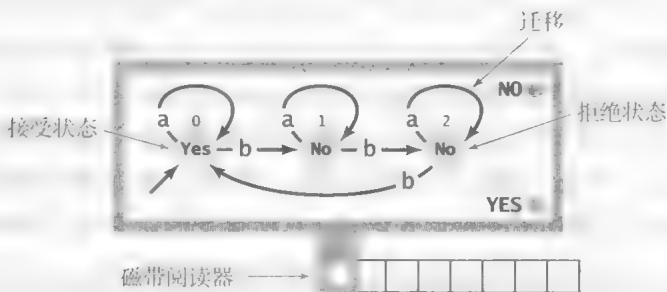
抽象机器是专用型的计算机。在本节中，我们专注于研究一个机器只解决一个问题的模型：即某种正则语言的识别问题。在适当的时候，我们将看到这个概念如何帮助我们了解通用计算机解决问题的原理。使用那些尽可能简单的机器，使得我们有可能通过严格证明来验证这些机器的行为。正如你将看到的，更重要且更令人惊讶的是，我们能够使用这些简单的模型解决适用性非常广泛的问题。

**确定有限状态自动机** 任何机器都需要能够读取一个输入符号，所以我们首先研究一种只处理数据输入的抽象机器，这种机器就是所谓的确定有限状态自动机（Deterministic Finite State Automaton, DFA）。每个 DFA 由以下部分组成：

- 一组数量有限的状态，每个状态都被指定为接受状态或者拒绝状态。
- 一组迁移，用于描述机器如何更改状态。对于符号集中的每个符号，每个状态都存在一个迁移。
- 一个纸带阅读器，这个阅读器最初位于输入字符串的第一个符号处，并且只能读取一个符号并移动到下一个符号处。

我们将每个 DFA 表示为一个图，其中接受状态为标记为“Yes”的顶点，拒绝状态为标记为“No”的顶点，并且每个迁移都是一条有向边，边上标记符号集中的一个符号。为了区分这些顶点，我们用从 0 开始的整数来对它们进行编号。这里有一个基于二进制符号集

的 DFA 的例子。这个 DFA 使用符号  $b$  和  $a$  作为符号集符号以避免与顶点的索引产生混淆。为了简化描述，如果我们有多个符号能够引起相同的迁移，我们将其表示为一条边，并在这条边上用一个包含了所有会引起这个迁移的符号组成的字符串来标记。



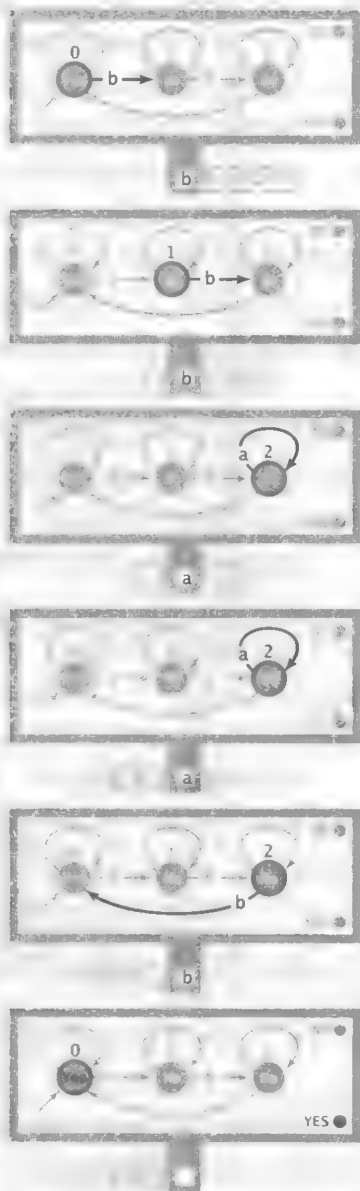
一个确定有限状态自动机

操作符。所有的 DFA 都从状态 0 开始，开始时它的一个输入字符串放在纸带上，此时磁头位于输入字符串中最左侧的符号上。机器根据下列规则以分步骤进行操作并同时改变状态：读取一个符号，将磁头右移一个位置，然后根据刚刚读取的输入字符的迁移标签来进行状态迁移。当输入全部读取完时，DFA 停止操作。如果停留的位置是一个接受状态，指示灯 Yes 亮起；如果该点是一个拒绝状态，指示灯 No 亮起。

例如对于我们示例的 DFA，给它一个输入字符串  $bbaab$  将有如下表现（如右图所示）：从状态 0 开始，它读取第一个  $b$  并移动到状态 1。然后读取第二个字符并移动到状态 2，因为那个符号是  $b$ 。然后它停留在状态 2，因为第三个和第四个符号都是  $a$ ，然后移动到状态 0，因为第五个符号是  $b$ 。当输入全部读取完毕时，机器处于 0 状态。状态 0 标记的是 Yes，所以它会激活 Yes 指示灯。换句话说，这台机器接受了二进制字符串  $bbaab$ 。通过这些步骤还可以看出，机器拒绝了二进制字符串  $b$ 、 $bb$ 、 $bba$  和  $bbaa$ 。

对于这些步骤的一个更紧凑的表示是 DFA 在尝试决定是接受还是拒绝这个二进制字符串时所生成的状态迁移序列，这个序列的最终状态表示这个二进制字符串是被接受还是拒绝。对于我们的例子，序列  $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 0$  就是这个 DFA 对于输入字符串  $bbaab$  的步骤序列。请注意，我们可以根据状态迁移序列来推断输入字符串。

描述语言的特征。哪些二进制字符串在一个给定的 DFA 所识别的语言中呢？为了回答这个问题，我们需要研究机器的行为来推断它接受和拒绝的字符串。在我们的例子中，你立刻就能看出来  $a$  的数量是不影响输出结果的，所以我们仅仅需要关心  $b$  的数量。这个机器拒绝  $b$  和  $bb$  但是接受  $bbb$ ；它拒绝  $bbbb$  和  $bbbbb$  但是接受  $bbbbbb$ （并且忽略了所有  $a$  的数量），



对一个 DFA 的操作进行跟踪



当我们说一个抽象机器的时候，意思是我们没有规定这个机器在物理世界中是如何实现的。这个做法让我们可以用不同的 DFA 来表示不同的情况。我们完全可以用一个数字表格，或者一个 Java 程序，或者一个电路，或者一块机械硬件来实现一个 DFA。事实上，DFA 适用于对可以在自然界中找到的所有物体进行建模。DFA 让我们可以对这些不同的机制的具体属性进行推理，无论是真实的机制还是抽象的机制。

**DFA 的 Java 实现** 那么我们如何建立 DFA 呢？事实上没必要这么做，因为开发一个被称为通用虚拟 DFA 的 Java 程序是非常简单的。通用意味着这个 DFA 可以模拟任何 DFA 的操作；虚拟意味着它只是在某台机器上作为一个程序实现，而不是一个物理设备。程序 5.1.3 是一个通用虚拟 DFA，它获取了一个 DFA 规范（从命令行参数指定其文件名）并且从标准输入获得一系列字符串，并且打印出给定字符串在 DFA 上运行的结果。

对于一个 DFA，文件的输入格式是：一个表示状态数目的数字后面跟着符号集，再后面跟着的每一行表示一个状态。对于每个状态所在的行包含了一个字符串，这个字符串首先是一个 Yes 或者 No（用于指定状态是否是一个接受状态），后面跟着的是对符号集中的每个符号在当前状态的迁移索引，第  $i$  个索引给出了当 DFA 输入符号集中的第  $i$  个符号时，从这个状态要如何进行迁移。程序下方显示的文件 b3.txt 定义了我们的 DFA，这个 DFA 识别的语言是 b 的数量是 3 的倍数。文件 gene.txt 定义了上述用于识别潜在基因的 DFA。

[741]

这个构造函数创建了这些数据结构，用于表示 DFA 的内部构造，而所需的数据由命令行参数指定的文件给出，获取信息的过程如下：

- 读取状态的数量以及符号集。
- 创建一个字符串数组，用于保存每个状态的接受 / 拒绝值，以及状态转换的符号表。
- 通过读取每个状态的接受 / 拒绝值和状态转换来填充这些数据结构。

实现这个构造函数的代码很简单：

```
public DFA(String filename)
{
 In in = new In(filename);
 int n = in.readInt();
 String alphabet = in.readString();
 action = new String[n];
 next = (ST<Character, Integer>[]) new ST[n];
 for (int state = 0; state < n; state++)
 {
 action[state] = in.readString();
 next[state] = new ST<Character, Integer>();
 for (int i = 0; i < alphabet.length(); i++)
 next[state].put(alphabet.charAt(i), in.readInt());
 }
}
```

程序 5.1.3 中的其他方法也很简单。simulate() 方法用于对 DFA 的操作进行模拟，DFA 中的 main() 方法对于每个来自输入的字符串调用 simulate() 方法。

程序 5.1.3 既是一个 DFA 构成的完整规范，也是一个不可缺少的研究特定 DFA 属性的工具。你提供符号集、一个 DFA 的表格式描述以及一系列输入字符串，它就会告诉你 DFA 是否接受这些字符串。它实际解决了 DFA 的识别问题，也简单展示了对虚拟机器的模拟，稍后我们还会讨论。它不是一个真正的计算设备，而是对这种设备是如何工作的一个完整定义。你可以将 DFA 视为一台“计算机”，通过描述一组顶点和迁移规则来进行“编程”，而

[742]

这些顶点的迁移条件的设置要符合 DFA 的规则。每个 DFA 都是这台计算机的一个“程序”。



程序5.1.3 通用虚拟DFA

```
public class DFA
{
 private String[] action;
 private ST<Character, Integer>[] next;
 public DFA(String filename)
 { /* 见上页 (432页) */ }

 public String simulate(String input)
 {
 int state = 0;
 for (int i = 0; i < input.length(); i++)
 state = next[state].get(input.charAt(i));
 return action[state];
 }

 public static void main(String[] args)
 {
 DFA dfa = new DFA(args[0]);
 while (StdIn.hasNextLine())
 StdOut.println(dfa.simulate(StdIn.readLine()));
 }
}
```

这个程序对于标准输入的每一行字符串，模拟了由命令行参数指定的DFA是如何完成识别过程的。如果字符串在指定的语言里面，这个程序打印Yes；否则打印No。

```
% more b3.txt
3
ab
Yes 0 1
No 1 2
No 2 0

% java DFA b3.txt
babbbabb
Yes
babbbb
No
```

```
% more gene.txt
11
ATCG
No 1 10 10 10
No 10 2 10 10
No 10 10 10 3
No 4 6 4 4
No 5 5 5 5
No 3 3 3 3
No 7 5 5 8
No 9 3 3 9
No 9 3 3 3
Yes 10 10 10 10
No 10 10 10 10
```

```
% java DFA gene.txt
ATGCTCTTTAG
Yes
ATGCCCTCTGA
No
ATGCCCTCCTTTCTAA
Yes
ATGTAA
Yes
ATGAAATAATAA
No
```

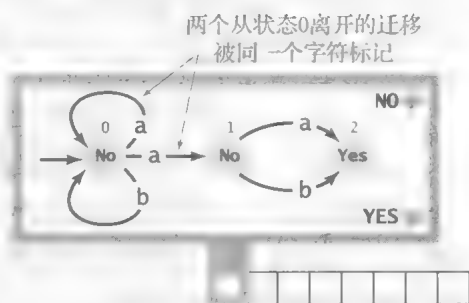
743

**不确定性** 接下来我们来考虑一个对 DFA 模型的扩展。这个模型与我们通常想象的在现实世界里的计算机有不同的思考方式。为什么我们要这么做呢？因为这个模型不仅可以创造更紧凑、更便于使用的自动机，还可以帮助我们创建更有用的实用程序，解决基础的理论问题。截至本书目前所述，它代表了一个 DFA 和正则表达式之间的链接，引领我们更充分地理解正则表达式，并实践正则表达式识别问题的解决方案。

**不确定有限状态自动机。**一个 DFA 的行为是确定 (deterministic) 的，即对于每个输入符号和每个状态都只有一个可能的状态迁移。一个不确定有限自动机 (NFA) 是一种有其他可能的状态迁移的抽象机器。具体来说，NFA 与 DFA 基本相同，只是取消了离开每个状态的转换限制，所以：

- 允许使用相同的符号来标记多个迁移。
- 允许未被标记的状态迁移 (空迁移) 存在。
- 并非所有符号都需要包含在每个状态的迁移中。

一个 NFA 不仅可以在没有读入输入符号的情况下进行状态迁移,而且在给定当前状态和输入符号的情况下它还可以有零个、一个或者多个状态迁移与每个状态和每个输入符号相对应。选择是否以及何时进行迁移并未指定。如果有任何一个迁移队列能让机器从起始状态(0)转换为接受状态,我们就认为这个输入是合法的。右图给出了一个 NFA 的例子。当机器处于状态 0 并读入一个 a 时,它可以选择不保持在状态 0 或者迁移到状态 1。接下来我们分析这个机器能识别哪种语言。

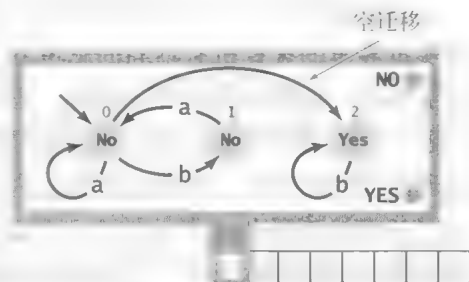


不确定有限自动机 (NFA)

**NFA 识别举例。**不同于有确定状态迁移规则的 DFA, 我们需要做更多的工作来解决 NFA 的识别问题。假设我们输入的字符串是 baaab: 如果你仔细研究一下, 你会发现 NFA 可以通过状态迁移序列 0-0-0-1-2 来接受这个输入。它也可以做像 0-1-2 或者 0-0-0-0 这样的状态迁移, 但是这些序列并不会用尽输入字符串并到达一个接受状态; 因此我们对这些转换不感兴趣。同时, 很容易看出, 如果输入的字符串是 bbbb, NFA 就没有办法从状态 0 迁移到状态 1, 所以它必须亮起 No 指示灯。为了解决 NFA 的识别问题, 我们必须考虑到所有的可能性。这种识别方法与 DFA 遵循简单地从一个状态到另一个状态的规则大相径庭。在这个例子中, 可以证明我们的示例 NFA 识别了所有倒数第二个符号为 a 的字符串的这种语言。首先, 通过下面的方法: 选择保持状态 0 直到读入倒数第二个字符串, 此时这个 NFA 读取 a 并进入状态 1, 然后在最后一个符号读取后进入状态 2 (接受状态), 这表示这个 NFA 可以接受倒数第二个符号是 a 的任意字符串。其次, 没有任何一个状态转换序列可以让这个 NFA 接受倒数第二个符号不是 a 的字符串, 因为输入符号 a 是到达状态 1 的唯一方法。

下图给出的是一个具有空迁移的 NFA (允许在不读入任何符号下进行状态迁移) 的例子。这让我们更难弄清一个特定的 NFA 是否能接受一个特定的字符串, 更不用说一个特定的 NFA 识别了哪种语言。图中的 NFA 识别的语言是不包含 bba 作为子字符串的所有二进制字符串的集合, 这可能需要花费一些时间来理解。我们将这种语言称为不存在 bba。

**NFA 的识别问题。**我们要如何解决 NFA 的识别问题呢 (即系统地检查一个给定的 NFA 是否接受一个给定的输入)? 对于 DFA 来说, 这个过程很简单, 因为每个步骤只有一种可能的状态迁移。然而对于 NFA 来说, 我们面临从大量可能性中找出至少一个状态迁移序列。



一个具有空迁移的 NFA

在我们定义 NFA 时, 我们不需要规定自动机如何进行计算; 事实上, 也很难想象我们要如何建立这样一台机器。对于每台机器这看起来都似乎是相当困难的工作, 因为一台机器不仅要为在它的语言里的那些输入找到一个能从起始状态转换到接受状态的迁移序列, 还要证明对于不在它的语言里的那些输入不会存在这样的一个序列。

一种思考 NFA 操作的方式是猜测一个可以到达接受状态的迁移序列: 如果有这样的一个序列, 那么它会找到这个序列。直觉告诉我们机器不能猜到它们计算的结果, 但是非确定性相当于我们想象它们可以这么做。在类 Java 的语言中我们需要一个类似于这样

的结构:

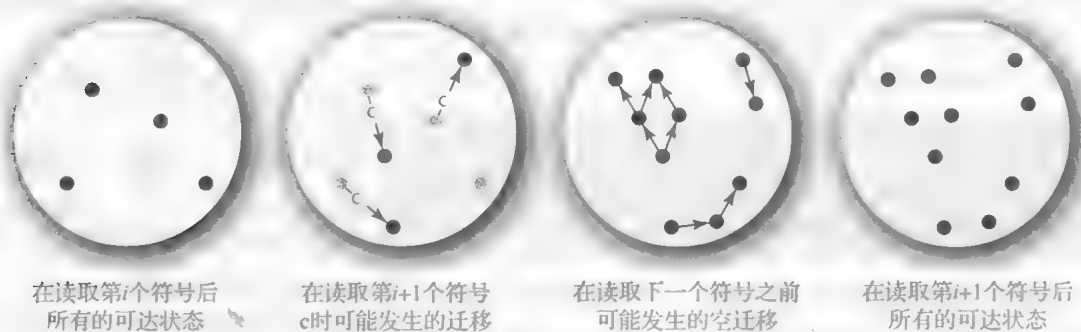
```
do either { this statement } or do { that statement };
```

这样导致看起来好像非确定性使我们远离现实世界。那么一个 Java 程序怎么决定要做什么呢? 实际上, 非确定性是一个既在实践中具有实用价值, 又在理解计算的基本性质方面至关重要的概念。

幸运的是, 按下面所示的方法来跟踪所有可能的状态转换, 以期模仿 NFA 的操作并不困难:

- 首先我们要找到所有可以从起始状态通过空迁移到达的状态。这是 NFA 在阅读第一个符号之前可以到达的所有状态的集合。
- 查找可以通过一条由第一个输入符号标记的边所到达的所有状态, 然后查找可以从这些状态中的其中一个通过空迁移到达的所有状态。这是 NFA 在阅读第二个符号之前可以达到的所有可能状态的集合。
- 重复这个过程, 跟踪 NFA 在读取每个输入符号之前可以到达的所有状态, 直到输入用尽。

在此过程结束时, 如果任何接受状态处于输入耗尽后留下的状态集合中, 那么 NFA 就会接受这个输入字符串在语言中的事实。如果没有任何接受状态处于输入耗尽后留下的状态集合中或者输入耗尽之前留下的状态集合为空, 则 NFA 会接受输入字符串不在语言中的事实。这些步骤是确定性的: 它们构成了一种无须任何猜测却能理解基于任何 NFA 的计算的方法。



模仿 NFA 操作中的一个步骤

746

**NFA 追踪示例。**在下面的例子中我们考虑如下情况, 我们给定一个输入 `ababb`, 对于“不出现 `bba`”这个 NFA 研究其迁移过程, 如下图所示。我们从状态集合  $\{0,2\}$  开始, 状态 0 是开始状态且有一个空迁移可以转换到状态 2。在读入第一位输入 (`a`) 之后可能到达的状态集合仍为  $\{0,2\}$ , 因为输入为 `a` 时仅有的一个迁移是从状态 0 回到状态 0, 然后也可能再以一个空迁移转换到状态 2。在读取第二位输入 (`b`) 之后, 我们可以处于状态 1 (从状态 0 经过 `b` 迁移得到) 或者处于状态 2 (从状态 2 经过 `b` 迁移回到自身)。输入的第三位是 `a`, 此时唯一的可用迁移是从状态集合  $\{1,2\}$  回到状态 0, 并且状态 0 总是可以通过一个空迁移转换到状态 2, 所以当我们读完第三位时仅有状态  $\{0,2\}$  是可能的结果。与前面一样, 在读完第四位后会得到一个 `b`, 将我们从状态  $\{0,2\}$  迁移到状态  $\{1,2\}$ , 然后在读完第五位之后, 字符 `b` 使得状态 2 自迁移到状态  $\{2\}$ 。既然这是已经耗尽了输入并处于状态 2 (一个接受状态),

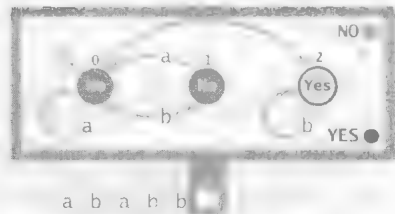
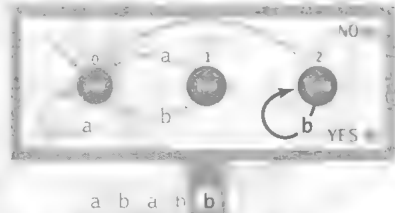
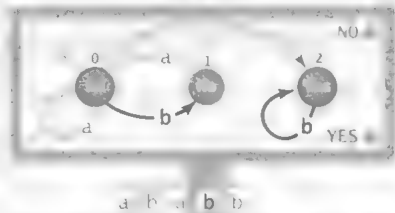
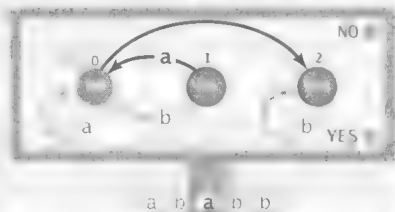
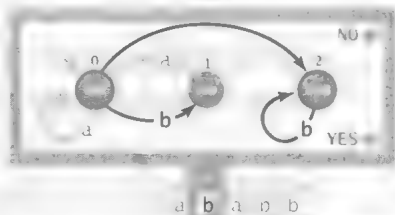
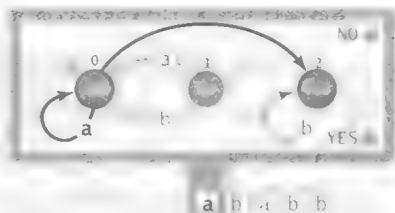
我们认为这个 NFA 将接受 `ababb` (而且我们可以展示出机器会做出的状态转换序列: `0-0-1-0-2-2-2`)。如果输入再加上一个额外的 `a`, 在读取完之后可能到达的状态集合会变成空 (因为在读取完第五位之后唯一的可能状态为状态 2, 并且状态 2 没有以 `a` 为输入的迁移), 所以 `ababba` 会被这个 NFA 拒绝。

对于 DFA 来说, 我们在程序 5.1.3 中通过一步一步模拟 DFA 操作的方式得到了一个 DFA 识别问题的解决方案。为了在 NFA 中做到同样的事情, 我们使用了与刚刚提到的跟踪过程相同的方法: 我们在程序计数器中跟踪了一组状态而不是跟踪单个状态, 这组状态是在读完纸带上当前的符号之后, NFA 可以到达的所有状态的集合。我们将这个方案的实现留在了练习题部分。让机器能猜出正确答案的想法是一个幻想, 但是能研究和使用非确定性的程序让这个想法变得像是真的一样。同样的, 我们可以将这个 Java 程序看成一个“NFA 计算机”, 并将每一个 NFA 看成这个计算机的一个“程序”。一台计算机被赋予对该做的事情进行猜测的能力, 它是否比 DFA 计算机——程序 5.1.3 中的通用虚拟 DFA 更强大? 存有这样的疑问是合理的。

**克林定理** 在正则表达式, DFA 和 NFA 之间有着惊人的联系, 这个联系具有重大的实践和理论意义。这个联系被称为克林定理 (Kleene's theorem), 它以在 1951 年建立了自动机和正则语言之间联系的逻辑学家史蒂芬·克林 (Stephen Kleene) 的名字来命名。

**定理:** (克林, 1951) 正则表达式、DFA 和 NFA 是相同的模型, 因为它们都可以表示正则语言。  
证明框架: 参见下面的讨论。

克林定理是我们要介绍的第一个主要定理, 所以有必要对一些数学证明的思路进行说明。在这个介绍中, 我们没有用标准的数学方式来陈述和证明定理。我们提出一个定理陈述的目的是为了让你意识到这是一个重要事实, 我们提出一个证明框架的目的是让你相信这个定理为真。我们会给出每个证明背后的意图以及足够让你理解这个证明的细节, 这样那些对这个证明感兴趣的人可以补全证明的剩余部分。我们的第一个证明是有建设性的, 也许这个证明比其他类型的证明 (比如那些更依赖于逻辑原理的证明方式) 更容易检查, 因为我们给出了一些例子来展示证明的过程。尽管示例并不比测试用例更能证明程序是有效的, 它们仍然是有必要的的第一步。如果你读了一个证明, 然后研究了一个例子, 再重读这个证明, 你会发现更容易让自己理解这个定理的真实性。虽然你比一个训练有素的数据家更容易被说服, 但也请你相信我们不会误导你。



跟踪 NFA 的操作

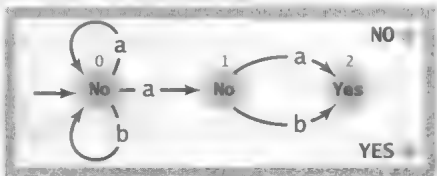
**证明策略。**根据定义，如果我们可以写出一个正则表达式来描述一个语言，那么这个语言是正则语言。任何正则表达式都描述了一个正则语言。为了证明克林定理，我们想要证明这个定义对于 DFA 和 NFA 同样如此。我们将分两步进行：首先我们证明 DFA 和 NFA 是等价的，然后我们证明 NFA 和正则表达式是等价的。

748

NFA 和 DFA 是等价的。每个 DFA 都是一个 NFA。如果我们对于任意一个 NFA 可以构建一个 DFA，其与给定的 NFA 具有相同的语言，那么就可以证明两者之间是等价的。首先，对于每一个可能的 NFA 状态集合，这个 DFA 都有一个状态与之对应。这意味着如果这个 NFA 有  $n$  个状态，那么这个 DFA 可能有  $2^n$  个状态。对于一个给定的 NFA，在它读取了输入中任意给定的部分之后，都有一个唯一可能的状态集合，所以这个 DFA 可以处理所有可能出现的情况。为了弄清楚转换过程，我们使用与我们讨论 NFA 的操作时相同的描述方法。对于一组给定的 NFA 状态集合和一个输入符号，我们从原集合中任意一个 NFA 状态出发，通过这个符号标记的边和任意数量的空边，从整个 NFA 状态集合中挑选出那些可达的 NFA 状态，组成一个新的 NFA 状态集合。这个 NFA 状态集合对应于我们正在构建的 DFA 中的单个状态，并且这个状态就是我们的迁移目标。对于 DFA 中的每个状态和符号集中的每个符号，我们都可以通过这种方式计算出结果。简便起见，我们可以忽略 DFA 中那些从状态 0 无法到达的状态。最后我们将每个与 NFA 中包含接受状态的状态集合相对应的 DFA 状态都定义为接受状态。

右边的例子展现了根据 NFA 构建 DFA 的过程。这个 NFA 的作用是识别倒数第二个字符为  $a$  的二进制字符串，它有三个状态，所以这个 DFA 应该被构造成拥有八个状态，每个都在图表中注明与哪个 NFA 状态对应：(空状态)，0，1，2 (单状态子集)，01，02<sup>⊖</sup>，12 (两个状态的子集) 和 012 (所有三个状态)。对于这些状态中的每一个，我们都可以为每个输入符号来决定它的结果。例如，在状态为 01 的时候给定一个输入符号  $a$ ，我们可以从 0-0、0-1 或者 1-2 这些 NFA 迁移中任意选择一个，所以对于进行  $a$  迁移的状态 01 在 DFA 中的后继者状态为 012；同样的，如果在状态为 01 的时候输入的符号是  $b$ ，我们可以从 0-0 或 1-2 这两个 NFA 迁移中任意选择一个，所以对于进行  $b$  迁移的状态 01 在 DFA 中的后一状态为 02。图中的转换表给出了需要用这种方式来分析的八种可能性。注意状态 1 和状态 12 没有作为任何一条边的目标状态出现过；状态 2 只能从这两个状态中的其中一个到达，状态  $\emptyset$  只能从状态 2 和自身到达，所以这些都是从

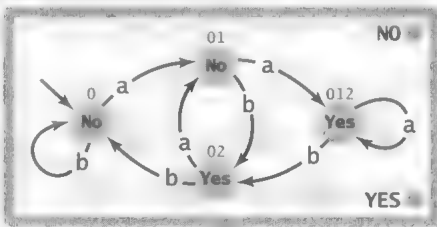
识别  $(alb)^*a(alb)$  的 NFA



NFA 到 DFA 的转换表

| 名称          | 接受  | a           | b           |
|-------------|-----|-------------|-------------|
| $\emptyset$ | No  | $\emptyset$ | $\emptyset$ |
| 0           | No  | 01          | 0           |
| 1           | No  | 2           | 2           |
| 2           | Yes | $\emptyset$ | $\emptyset$ |
| 01          | No  | 012         | 02          |
| 02          | Yes | 01          | 0           |
| 12          | Yes | 2           | 2           |
| 012         | Yes | 012         | 02          |

识别  $(alb)^*a(alb)$  的 DFA



749

从一个 NFA 转换为一个 DFA

⊖ 原书错误。——译者注

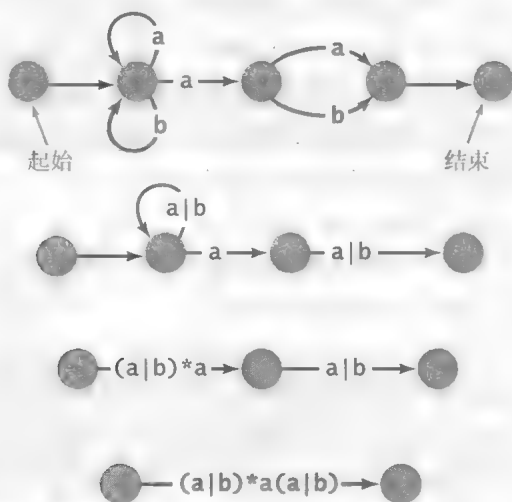
状态 0 无法到达的状态，我们可以在 DFA 中忽略它们。

为了相信 DFA 和 NFA 是等价的，你需要确信原则上对于每个 NFA 你都可以完成以上过程（即使在某些情况下可能因为状态太多而令人感到乏味）。你可以在练习中找到其他例子。注意，在有空转换的 NFA 中，你需要做的第一件事就是计算起始状态（见练习 5.1.14）。

简而言之，没有任何形式语言是可以被某些 NFA 识别但是不能被 DFA 识别的。你可能会有一个先入为主的想法，即：NFA 机器会比 DFA 功能更强大，但事实并非如此。它们有着相同的计算能力。

NFA 与正则表达式是等价的。为了构建一个正则表达式，其能够描述被任何给定的 NFA 识别的语言，我们可以进行以下步骤：

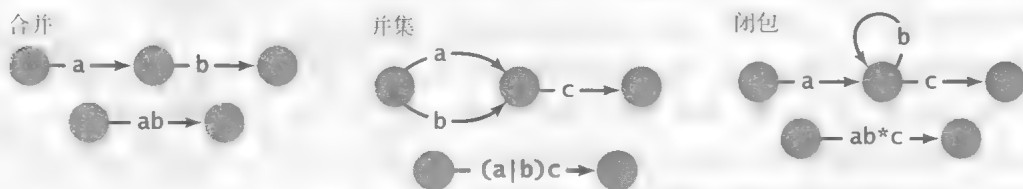
- 扩展 nNFA 模型，以使得边上的标记可以是正则表达式。
- 设置一个起始状态，以使其空边可以到达 NFA 的起始状态；设置一个结束状态，以使从每个接受状态都可以经过空边到达它。
- 一次从 NFA 中删除一个状态，把这个状态所代表的功能用更复杂的正则表达式和相应的边代替，直到只剩下起始状态和结束状态，这两个状态之间通过一条用正则表达式标记的边相连接。



将一个 NFA 转化为一个正则表达式

我们用来移除状态的三个基本操作在下面说明。为了简单起见，例子上用的都是单符号标记的边；换成正则表达式同样也是有效的。而且它们没有考虑到从其他边进入和退出顶点来移除状态的可能性。但是我们总是可以分四步以从任何扩展的 NFA 移动任意状态  $x$ ：

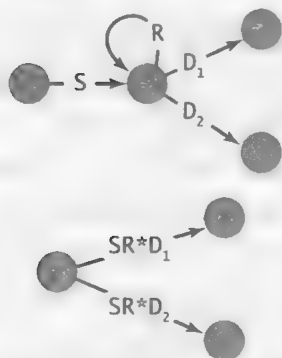
[750]



将一个顶点从一个扩展 NFA 中删除的三种方式

- 用 “|” 结合从相同来源进入  $x$  的边。
- 用 “|” 结合从  $x$  离开并且到相同目的地的边。

- 前序操作中离开  $x$  的操作最多有一个自循环 (将它标记为  $R$ )。对于每一对出 / 入该状态的边 (将它们分别标记为  $S$  和  $D$ )，创造一条标记为  $SR^*D$  的新边，这条边的来源为原入边的来源，目的地为原出边的目的地 (这样就能跳过  $x$ )。
- 删除  $x$  和所有进入及离开它的边。



将一个存在两条出边的 NFA 节点删除

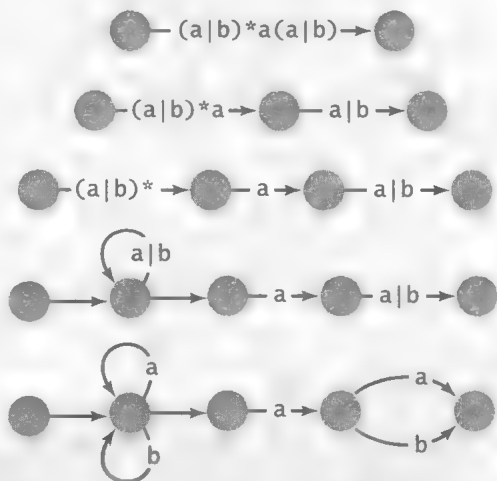
左图是这个基本操作的一个例子。我们很容易就能验证这些操作会产生一个与原始 NFA 等价的 NFA。新的 NFA 可能会比原始 NFA 有更多的边 (当一个顶点有  $m$  条入边和  $n$  条出边时，我们用  $m*n$  条边来代替这个顶点和那些边)，更少的状态。因此我们可以应用这些操作直到没有状态剩下 (仅仅剩下一条边)。在这条边上的标签就是能描述被这个 NFA 识别的语言的正规表达式。前面的示例展示了将一个 NFA 转换为一个正规表达式的步骤，这个 NFA 规定的是倒数第二个符号为  $a$  的二进制字符串。这个过程足够详细，我们可以编写 Java 程序来完成这个转换，但是在这里我们关心的仅仅是证明：你可以将任何一个 NFA 转换为一个正规表达式，用来描述这个 NFA 识别的语言。

如果把相同的步骤反过来，我们就可以构建一个 NFA，这个 NFA 可以识别与给定正规表达式描述相同的语言。我们从一个简单的只有一条用正规表达式标记的边的 NFA 开始工作，通过添加状态来化简边上的正规表达式直到每条边上的标签都为  $a$ 、 $b$  或者没有标签，从而构建一个 NFA。我们从一个单状态的扩展 NFA 开始，这个 NFA 只有一条由给定正规表达式标记的入边和出边。现在，每个拥有特殊标签的边必须满足  $(R)$ 、 $RS$ 、 $R|S$  或者  $R^*$  中的一种 ( $R$  和  $S$  都是正规表达式)，因此我们可以按照下面任一方式进行简化：

- 如果一条边标记为  $(R)$ ，将括号删除。
- 如果一条边标记为  $RS$ ，将这条边替换为一个新的状态和两条新的边，一条从源状态到新状态的边标记为  $R$ ，另一条从新状态到目的地的边标记为  $S$ 。
- 如果一条边标记为  $R|S$ ，则用平行的两条边代替这条边，其中一条标记为  $R$ ，另一条标记为  $S$ 。
- 如果一条边标记为  $R^*$ ，则用没有标签的边和标记为  $R$  的自循环边来代替这条边。

这些构造方法都会生成一个扩展 NFA，这个扩展 NFA 能识别与源 NFA 相同的语言并且简化了一些边，所以我们可以继续使用以上构造方法直到每一条边都标记为空、 $a$  或者  $b$ ，这样就得到了一个 NFA。右图描绘了这种构造方法的一个例子。其与将一个 NFA 转换为一个正规表达式的过程基本相同。同样的，你需要做一些在本节结尾部分的练习来让你自己相信，可以用这种方法来对任意一个给定的正规表达式建立一个 NFA。

我们已经确定了 DFA、NFA 和正规表达式是等价的 (克林定理)。任何可以被正规表达式、



将一个正规表达式转换为一个 NFA



DFA 和 NFA 描述的语言都是正则的，任何一个正则语言都可以被正则表达式、DFA 和 NFA 描述。我们详细地描述这个定理的证明，可以让你了解到抽象机器的这种思维，即使是最简单的机器也可能带来令人惊讶的理论结果（非确定性并不会增强 DFA 的能力），并向你介绍一个有趣又有用、好理解的计算模型，以及向你描述一种简单有效的基本方法，这些方法可以用来支持一些计算理论中基础并且深刻的思想。

我们还有一件需要注意的事：我们一直在关注做这些迁移的可能性；但是做这些事的成本完全就是另一回事。例如，与一个具有  $n$  个状态的 NFA 对应的 DFA 可能有  $2^n$  个状态。事实上，在我们的例子中，我们都知道没有一个少于  $2^n - 1$  个状态的 DFA 可以识别“在最后  $n$  位只有一个 a 的二进制字符串”这种语言。当  $n$  较大时，这个代价是过高而且无法实施的。我们现在可以知道正则表达式、DFA 和 NFA 可以做什么，但实际应用中我们需要将成本也一并考虑。我们会在 5.5 节中回到这个话题。

[752]

**克林定理的应用** 克林定理既是一个重要的理论成果，也可以直接应用于实践。接下来我们会介绍两个克林定理的重要应用，一个例子从理论出发，另一个从实践出发。

**正则表达式识别。**这个理论的基本实践应用是作为解决正则表达式识别问题的基础：给定一个正则表达式和一个字符串，这个字符串在被正则表达式描述的语言中吗？我们对克林定理的证明是有建设性的，所以它为构建一个解决正则表达式的识别问题的程序提供了一个直接的途径：

- 构建一个与给定正则表达式相对应的 NFA。
- 在给定的输入字符串上模拟 NFA 的操作。

这就是前面一节我们讲到的，Java 实现 `match()` 方法时首先会采取的方案。这个实现需要特别注意确保计算的代价在可控范围内。你可以在我们写作的另一本书《算法》（第 4 版）中找到完整的描述和实现。下例是一个计算机科学领域的基础范例：

- 选择一个中间抽象概念（在这个例子中为 NFA）。
- 建立一个模拟器来使抽象概念具体化。
- 构建一个编译器，来将问题转换为抽象概念。

这个过程就是将 Java 程序编译成 Java 虚拟机所需要的程序的过程（这些 Java 虚拟机程序本身也是一个抽象概念，并且最终用那些为特定计算而编写的低级语言程序来模拟）。

**DFA 能力的限制。**克林定理还帮助我们揭示了一个基本的理论问题：哪种形式语言可以被一个正则表达式描述，以及哪些不能？

要解决这个问题，我们首先考虑“所有 a 的数量和 b 的数量相等的字符串”这种语言。至少这种语言看起来与我们在本节介绍过的许多其他语言一样简单。我们可以写出一个描述这个语言的正则表达式吗？我们当然可以开动自己的想象试试看，但事实上这种语言是非正则的，所以没有办法用一个正则表达式来描述它。

我们怎样才能证明这样一个结果呢？克林定理使得我们可以不用对正则表达式表达事物进行直接推理，而是用抽象机器代替。由克林定理建立的正则表达式、DFA 和 NFA 的等价关系意味着，我们可以使用它们之中的任意一个来表示一个正则语言的概念。在这个例子中，我们使用 DFA。

[753]

**论断：**不是所有的形式语言都是正则的。

**证明：**采用反证法，我们首先假设“a 的数量与 b 的数量相等”这种语言是正则

的——我们可以写出描述它的正则表达式。由克林定理知，我们可以建立一个能识别这种语言的 DFA。设 DFA 中的可以接受“a 的数量与 b 的数量相等”的状态数量为  $n$ 。具体来说，它能识别一个字符串，这个字符串由一个有  $n$  个 a 符号的序列和一个有  $n$  个 b 符号的序列组成。现在让我们来研究一下这个 DFA 在接受这个字符串时它访问的状态的轨迹。因为 DFA 只有  $n$  个状态，所以由抽屉原理 (pigeonhole principle) 知，这个 DFA 在读取 a 时必须重复访问一个或多个状态，因为在轨迹中有  $n+1$  个状态 (含开始状态)。例如当  $n$  为 10 时，轨迹可能如下所示：

```
a a a a a a a a a b b b b b b b b
0 3 5 2 9 7 3 5 2 9 7 1 5 4 2 9 6 8 7 8 7
```

在这个例子中，状态 3 会在第六次转换时重复。现在，我们可以通过从原始字符串中在每对重复状态之间去除相应的 a 来构造不同的输入字符串。在上面的例子中，我们将去除 3-5-2-9-7-3 中的 5 个 a 字符串，然后我们可以通过减少轨迹来得到这个字符串的轨迹，如下所示：

```
a a a a a b b b b b b b b b
0-3-5-2-9-7-1-5-4-2-9-6-8-7-8-7
```

观察得到的关键结果是，DFA 的第二次输入和第一次输入一样以相同的最终状态结束。因此它也接受了第二个输入。但是这个输入中 a 的数量比 b 要少，这与我们所做的假设“DFA 能识别这个语言”相矛盾。这个假设是从我们存在一个可以描述这种语言的正则表达式得来的，因此，我们证明了没有正则表达式可以描述“a 的数量与 b 的数量相等”这个语言。

这个证明有两个值得你关注的证明技巧。第一个是反证法 (proof by contradiction)，即为了证明一个陈述是真的，我们首先假设它是假的。然后我们逐步进行逻辑推理得出一个错误的结论。为了使这个结论成立，原假设必须是错误的；也就是说，这个陈述必须是真的。第二个证明技巧是抽屉原理：假设有  $n+1$  只鸽子进入  $n$  个或更少的鸽子洞，那么必须有一些鸽子洞是有多于一只鸽子在里面的。如果你缺少进行数学证明的经验，你应该带着这些技巧重新阅读一遍证明过程。

754

仅仅展示一个非正则语言不足以完成整个证明，但是这个证明的技巧适用于许多其他语言，并且能帮助我们理解正则语言的特征是什么。许多有用的简单语言都是非正则的。

功能更强大的机器。更重要的是，非正则语言带给我们另一个关键问题：将抽象机器模型扩展成可以识别这种简单语言的机器的最简单方法是什么？根据克林定理，即使是具备非确定性功能也无济于事。当我们找到这么一个模型时，我们首先需要了解这个模型中的机器能够识别的语言集合的边界在哪里、极限在哪里。

有限状态自动机的根本限制是它们的状态是有限的，所以它们能跟踪的事物的数量是有限的。在上面的例子中，自动机无法辨别有 10 个 a 字符的输入字符串和只有 5 个 a 的输入字符串之间的差异。一个简单的克服这个缺陷的方法是给 DFA 增加一个下推栈，得到一个下推自动机 (Pushdown Automaton, PDA)。通过将多余的符号保存在栈里，我们不难构建一个可以识别 a 和 b 的数量相等这种语言的 PDA (见练习 5.1.44)。有一种语言被称为上下文无关语言 (context-free language)，这种语言简单、易于理解且非常有用，它与 PDA 能识别的那些语言相似。例如，所有可能的正则表达式都是一个上下文无关语言，同时也是 Java

755 的一个核心子集。

有没有不能被 PDA 识别的语言呢？有没有比 PDA 更强大的自动机呢？是的，这样的语言和机器是存在的。你可能会期望有一个长长的机器模型的列表，列表上的每个机器都比上一个机器有更强大的功能，但是这个列表实际上很短。事实上，我们将在下一节看到，给机器增加第二个栈就可以使得机器像任何人想象的那样强大。

**总结** 正则表达式、DFA 和 NFA 是用于描述被称为正则语言的一些语言的等效模型。这种关系让我们可以证明正则语言的一些特性，并开发一些程序，利用它们的特性来完成各种基本计算任务。

现实中存在很多非正则但是十分有趣的语言——哪种形式化的机制可以定义它们呢？是否有不能用这些更普遍的机制来描述的语言呢？这些类型的问题和语言之间的关系、用于描述它们的形式化方法，还有通过克林定理演示的抽象机器，都为我们在本章开头提出的问题提供了更丰富的相关知识：

- 有些计算机本质上就比其他计算机更强大吗？
- 哪些问题可以用一台计算机解决呢？
- 计算机的极限在哪？

在本节中，我们彻底解决了有限自动机器的这些问题，通过这些问题我们可以精确定义简单的抽象计算机。在解决这些问题时，我们考虑了大量这些简单抽象模型的实际应用情况，而且我们又增加了两个基本问题：

- 计算机和语言之间的关系是什么？
- 非确定性会使机器变得更强大吗？

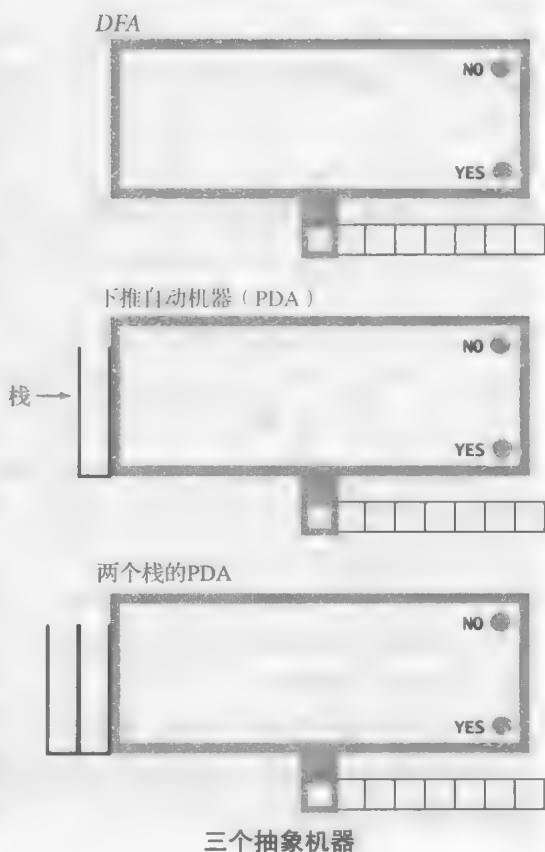
接下来，我们会通过考虑这些问题来获得更强大的计算模型。我们需要考虑的模型只比 DFA 和 NFA 稍微复杂一些，但是它们适合所有已知的计算设备。这些计算模型构成了计算机科学的核心。

756

## 问答环节

问：符号的定义是什么？

答：符号是形式语言的基本抽象构件。从一个数学家的角度来看，符号的定义并没有什么区别：对形式语言的研究就是对符号序列集合的研究。从科学家或工程师的角度看，符号是至关重要的，因为它们构成了形式语言抽象世界与我们使用形式语言的真实世界之间微弱的联系。一个符号可能是真实世界中的任何一个东西，比如一个触发器、一个基因、一个神经元、一个分子，或者是抽象世界中的任何一个东西，比如一位、一个数字、一个小写字母



或者一个 Unicode 符号。

问：交集也是集合的基本操作之一。为什么不把交集也作为正则表达式基本定义的一部分呢？

答：这是个好问题。由于语言仅仅是一些集合，所以很多东西可以从集合理论延续到正则表达式中。你可以为语言定义交集，而且两个正则语言的交集语言也是正则的（见练习 5.1.37）。

问：那些用于指定元符号的转义字符有点令人困惑。可以使用缩写来描述它们吗？

答：恰恰相反，只会写得更长。当你在 Java 字符串文本里描述一个正则表达式时，你需要另一个级别的转义来转义字符元符号。例如，你需要输入 “\s” 来代表一个空格字符，输入 “\\” 来代表一个单斜杠字符。有时候你需要对命令行中的特殊字符进行转义。例如，美国货币的正则表达式，它的命令行参数使用了 “\\\$” 来表示一个美元符号，使用 “\\.” 来表示一个小数点。

问：学习 NFA 有什么意义？它们比 DFA 更难想象，我不清楚它的优势在哪。

答：我们使用 NFA 来证明克林定理，并且使用它们可能实现正则表达式识别问题。另一个原因是 NFA 往往比 DFA 拥有更少的状态。尝试为倒数第 10 个符号为 1 的所有字符串构建一个 DFA。你会发现即使构建这种语言的 NFA 仅仅需要 10 种状态，却无法在 512 种状态内构建一个 DFA。

757

问：如何才能写出一个正则表达式来指定包含空字符串的单元元素集合？

答：我们使用符号  $\epsilon$  来表示包含长度为 0 的字符串的单元元素集合。我们还使用  $\emptyset$  来表示不包含任何字符串的空集。严格来说，这些都应该包含在我们对正则表达式的正式定义中。

问：空集的闭包是什么？

答：空字符串： $\{\}^* = \epsilon$ 。

758

## 练习

5.1.1 编写一个解决回文识别问题的 Java 程序，从标准输入中获取输入字符串序列。为了简单起见，我们假设符号集是罗马字母。具体来说，对于标准输入中的每个字符串，如果字符串是回文，则程序打印这个字符串且后面接 “is a palindrome”，如果字符串不是回文，程序应该打印这个字符串且后面接 “is not a palindrome”。

5.1.2 以练习 5.1.1 的方式，编写一个解决 “具有相同数量的 a 和 b” 这种语言识别问题的 Java 程序。我们假设标准输入上只会有 a 和 b 出现。

5.1.3 证明：前文 “确定有限状态自动机” 部分的 DFA 能识别 “b 的数量是 3 的倍数” 这种语言。

5.1.4 基于符号集  $\{a,b\}$ ，对于下列正则表达式描述的语言，给出一个简洁的文字描述：

a.  $a^*$

b.  $a^* a | a$

c.  $a^* abbabba a^*$

d.  $a^* a a^* a a^* a a^*$

5.1.5 基于符号集  $\{a,b\}$ ，对于以下每种语言，给出一个能描述它们的正则表达式。

a. 除空字符串以外所有的字符串。

b. 包含至少三个连续的 b。

c. 以 a 开头且长度为奇数或者以 b 开头长度为偶数。

d. 没有连续的 b。

e. 除了 bb 和 bbb 以外的所有字符串。

f. 以相同的符号开头和结尾。

g. 包含至少两个 a 和至多一个 b。

5.1.6 写出一个 Java 正则表达式, 这个表达式能匹配所有仅包含 5 个元音字母 (a、e、i、o、u) 并且字母按照这个顺序出现的单词 (如 *abstemious* 和 *facetious*)。

5.1.7 为 “July 4, 1776” 这样格式的日期写一个 Java 正则表达式, 这个正则表达式要尽可能的涵盖你认为合法的日期并且不能包含其他字符串。

5.1.8 为十进制数写一个 Java 正则表达式。一个十进制数即前面一串数字序列, 然后跟着一个小数点, 后面再跟一串数字序列。在这两个数字序列中, 至少有一个是非空的。如果第一个数字序列有不止一个数字, 那么这个序列不能以 0 开始。

5.1.9 在 Java 中为 (科学计数法的) 浮点数写一个 Java 正则表达式。一个浮点数以一个十进制数字为开头 (见前面的练习), 后面跟 e 或者 E 中的一个, 再跟 “+” 或者 “-” 其中的一个, 最后跟一个整数尾数。

5.1.10 定义 L 为语言 {aaaba, aabaa, abbba, ababa, aaaaa}。挑选出可以识别以下语言的正则表达式: (i) 不含 L 中的任意字符串; (ii) 含有 L 中的一些字符串且含有一些其他字符串; (iii) 含有 L 中的所有字符串且含有一些其他字符串; (iv) 与 L 完全相同。

a.  $a(a|b)^*abb(a|b)^*$

b.  $a(a|b)^*a$

c.  $a^*b^*aba$

d.  $a((a^*|b^*)|(b^*aba^*))a$

e.  $a^*b^*aa^*b^*ba^*a^*b^*b^*a^*a^*b^*$

f.  $(a|b)(a|b)(a|b)(a|b)a$

g.  $(a|aa|aaa)(ba|aa|bbb)a$

5.1.11 设计一个可以识别含有奇数个 a 和偶数个 b 这种语言的 DFA, 然后按照 DFA (程序 5.1.3) 期望的格式创建一个文件并测试你的设计。

5.1.12 设计一个可以识别社会安全号码的 DFA, 然后按照 DFA (程序 5.1.3) 期望的格式设计一个文件并测试你的设计。

5.1.13 写一个 Java 程序来解决被正则表达式  $C.\{2,4\}C\dots[LIVMFYWCX].\{8\}H.\{3,5\}H$  描述的语言的识别问题。这个程序不能使用 Java 自带的正则表达式机制, 以标准输入上用户输入的一系列字符串作为程序的输入数据。

5.1.14 设计一个可以识别语言  $a^*a|b^*b$  的 DFA, 然后设计一个可以识别相同语言且拥有更少状态的 NFA。

5.1.15 设计一个识别语言  $.^*aabab.^*$  的 DFA (所有含有 aabab 的字符串的集合), 然后设计一个可以识别相同语言且拥有更少状态的 NFA。

5.1.16 考虑这么一个自动售货机, 对于价值 25 美分的货物, 它可以接受 5 美分、10 美分, 以及 25 美分的硬币。设计一个对于每种可能的投入金额都有相应状态的 DFA, 并且添加迁移使得当投入的金额总数是  $i$  的 5 倍时, 机器处于状态  $i$ 。

5.1.17 描述如何将任意一个 DFA 转换为可以识别原来 DFA 识别的语言的补集 (基于相同的符号集, 不包含在该语言中的所有字符串的集合) 的 DFA。

5.1.18 将上一题中的 DFA 改为 NFA。

5.1.19 对于 “所有拥有奇数个 a 和偶数个 b 的二进制字符串” 这种语言, 创建一个可以识别它的 DFA。

5.1.20 将上一题中的 DFA 改为一个正则表达式。

5.1.21 画出对应以下正则表达式的 NFA:

a.  $a(a|b)^*a$

b.  $a^*b^*aba$

- c.  $(a|b)(a|b)a$  d.  $a((a^*|b^*)(b^*aba^*))^*a$   
e.  $ab(a|b)ba \mid a(a|b)aba \mid aa(ab|ba|aa)a$

5.1.22 将“NFA 识别举例”部分给出的 NFA (这个 NFA 能识别不包含字符串 bba 的字符串) 转换为一个 DFA。

5.1.23 是否有可能构造一个正则表达式来描述所有拥有相等数量的  $ab$  和  $ba$  的二进制字符串呢？  
答案：回答是肯定的， $(a.*a) | (b.*b) | a^*|b^*$  就是一个例子。虽然 DFA 不能够对  $ab$  和  $ba$  进行计数，但是在这种情况下，这个语言与“所有以相同位开始和结束的二进制字符串”语言相同，所以不需要计数。

**5.1.24** 证明没有 DFA 可以识别所有回文的集合。

5.1.25 证明以下是正则语言：不包含字符串 bba 并且 b 的数量是 3 的倍数。

761

## 创新练习

5.1.26 正则表达式挑战。下列每一个语言都是基于符号集  $\{0,1\}$  的二进制字符串，请你为这些语言分别构建一个正则表达式。

- a. 除了 11 和 111 之外的所有二进制字符串。
- b. 每个奇数位位置都是 1 的二进制字符串。
- c. 至少有两个 0 且至多有一个 1 的二进制字符串。
- d. 没有两个连续的 1 的二进制字符串。

5.1.27 二进制数的整除。对于下列每个选项，构建一个正则表达式以描述二进制字符串，当这些二进制字符串被当作整数来看时，可以使得选项描述的条件成立。

- a. 可以被 2 整除。      b. 可以被 3 整除。      c. 可以被 6 整除。

5.1.28 正则表达式搜索。检查你的计算机上的各种程序（网页浏览器、文字处理器、音乐库或者其他常用的软件），并确定它们的搜索功能是否支持正则表达式，以及能支持到什么程度。

5.1.29 可以被 3 整除 (二进制)。设计一个 DFA, 这个 DFA 可以识别的语言为所有可以被 3 整除的数字转换成的二进制字符串。例如, 1111 表示的是十进制 15, 所以可以被接受; 但是 1110 会被拒绝。提示: 使用三种状态, 你的 DFA 需要根据输入数字除以 3 得到的余数的不同来确定它应该处于三种状态中的哪一种。

5.1.30 弹跳过滤。有限状态转换器 (finite state transducer) 也是一种 DFA, 只是它的输出是一个符号序列。它大致与 DFA 相同, 只是它的每个迁移都标有一个输出符号, 每当进行迁移的时候都会输出这个符号。开发一个转换器用来移除输入中孤立的符号。具体来说, 输入中出现的任何 aaba 都应该在输出中被 aaaa 取代, 输入中出现的任何 bbab 都应该在输出中被 bbbb 取代, 否则输入和输出应该保持不变。

762

5.1.31 采集器。一个 Java 的 Pattern 对象代表了一个正则表达式。这样的一个对象可以用来为一个给定的字符串构建 Matcher 对象。Matcher 对象可以表示一个与这个正则表达式相对应的 NFA。Matcher 对象包括的操作有 find(), 它用于寻找字符串中下一个与正则表达式匹配的项; group(), 它用于返回能导致这种匹配的字符串字符。编写一个 Pattern 和 Matcher 的客户程序, 客户程序将一个文件名 (或者一个 URL) 和一个正则表达式作为命令行输入, 并且输出文件中所有与正则表达式匹配的子字符串。

答案:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class Harvester
{
 public static void main(String[] args)
 {
 In in = new In(args[1]);
 String re = args[0];
 String input = in.readAll();
 Pattern pattern = Pattern.compile(re);
 Matcher matcher = pattern.matcher(input);
 while (matcher.find())
 System.out.println(matcher.group());
 }
}
```

- 5.1.32 计数匹配。开发一个程序，这个程序可以对标准输入中与命令行上的正则表达式相匹配的子字符串进行计数，且不用担心重复匹配（使用前面练习中编写的 `Pattern` 和 `Matcher`）。
- 5.1.33 网络爬虫。修改练习 5.1.31 的解决方案来开发一个程序，这个程序可以打印出作为命令行参数的网页所能访问的所有网页。
- 5.1.34 一级正则表达式。构建一个 Java 正则表达式，这个表达式可以指定符合二进制正则表达式的字符串集合，但不会出现嵌套的括号。例如  $(a.*b)^*(b.*a)^*$  在语言中，而  $(b(a|b)b)^*$  不在语言中。
- 5.1.35 查找和替换。编写一个过滤器 `SearchAndReplace.java`，这个过滤器可以将一个正则表达式和一个字符串 `str` 作为命令行参数，从标准输入中读取一个字符串，将标准输入中的所有与正则表达式匹配的子字符串替换为 `str`，然后将结果发送到标准输出。首先使用 Java 的 `String` 库中的 `replaceAll()` 方法实现这个功能，然后不使用这个方法再给出一个解决方案。
- 5.1.36 空语言测试。在 DFA 中加入一个 `isEmpty()` 方法（程序 5.1.3），如果这个自动机能够识别的语言为空，那么返回 `true`，否则返回 `false`。

答案：这段代码使用了“广度优先搜索”和一个 `Queue` 来跟踪从状态 0 可以到达的状态（见程序 4.5.4）。

```
public boolean isEmpty()
{
 Queue<Integer> queue = new Queue<Integer>();
 boolean[] reachable = new boolean[n];
 queue.enqueue(0);
 while (!queue.isEmpty())
 {
 int state = queue.dequeue();
 if (action[state].equals("Yes")) return false;
 reachable[state] = true;
 for (int i = 0; i < alphabet.length(); i++)
 {
 int st = next[state].get(alphabet.charAt(i));
 if (!reachable[st]) queue.enqueue(st);
 }
 }
 return true;
}
```

- 5.1.37 为语言设置操作。给定可以识别两种语言 A 和 B 的 NFA，思考一下如何构造可以识别 A 和 B 的并集和交集的 NFA。



- 5.1.38 随机输入。对于一个给定的 DFA，编写程序来估算它能识别一个随机生成的长度为  $n$  的二进制字符串的可能性。
- 5.1.39 NFA 到 DFA 的转换。编写一个程序，这个程序可以从一个 NFA 中读取它的描述，并创建一个能与这个 NFA 识别相同语言的 DFA。
- 5.1.40 最小的语言。给定一个正则语言  $L$ ，证明  $L$  中最小字符串的集合也是正则的。最小字符串的意思是，如果字符串  $x$  在  $L$  中，那么对于任何一个字符串  $xy$  ( $y$  为任一非空字符串) 一定不在  $L$  中。
- 答案：对 DFA 进行修改，使得一旦它离开接受状态，就总是处于拒绝状态。
- 5.1.41 反向引用。证明反向引用操作是一个“不那么正则”的表达式，因而不能以核心正则表达式操作来构造。证明没有 DFA 可以识别  $ww$  形式的语言，其中  $w$  是任意字符串（例如 *beriberi* 和 *couscous*），但是 Java 正则表达式  $(.*)1$  可以描述这种语言。
- 5.1.42 通用虚拟 NFA。开发一个类似程序 5.1.3 的程序，这个程序可以模拟任何 NFA 的操作。像练习 5.1.36 那样，使用一个 NFA 的图形表示和一个跟踪它可能状态集合的 Queue 来实现本文中描述的方法。对程序进行测试，令程序在读取每个符号之前打印 NFA 所有可能状态的集合。测试你的代码，其中 NFA 可以识别所有倒数第四个符号是一个 1 的字符串组成的集合，输入数据是 *aaaaababaabbbbaababbbb*。
- 5.1.43 通用虚拟 PDA。一个下推自动机与 DFA 相比增加了一个堆栈，以及在任何迁移时将符号压入栈中或从栈中弹出符号的能力。具体来说，每个迁移都有一个额外的标签，如果标签是  $x/y$ ，则弹出堆栈，并且当弹出符号是  $x$  时压入  $y$ ；如果标签是  $/y$ ，则表示仅仅将  $y$  压入而不弹出任何符号；如果标签是  $/x$ ，则意味着弹出堆栈，并且当弹出符号是  $x$  时不压入任何符号；如果标签是  $/$  则意味着仅仅弹出堆栈。开发一个类似于程序 5.1.3 且可以模拟任何 PDA 操作的程序。
- 5.1.44 非正则语言的 PDA。画出一个可以识别有相等数量的  $a$  和  $b$  语言的 PDA（见之前的练习），并且用前面练习的解决方案来测试它。

765

## 5.2 图灵机

艾伦·图灵在 1937 年发表的论文中介绍了 20 世纪最美丽而有趣的知识发现之一，这是一个简单的计算模型，它足以体现任何计算机程序的特征，并且构成了计算机理论的基础。因为它的描述和行为都很简单，所有我们可以很容易地对它进行数学分析。这个分析让图灵对计算机和计算有了更深入的理解：所有已知的计算设备从深层次的技术原理层面来说都是等价的，而且有一些计算问题无论处理器有多快或者有多少可用的内存，都是根本无法解决的。我们会在接下来的两节中讨论这些想法，作为基础，我们首先需要分析图灵的发明到底是什么，以及它的概念和基本属性是什么。

如果你在其他地方曾经阅读过有关图灵机的资料，或者将来在其他材料中遇到图灵机，你会发现它的定义和某些属性与我们给出的略有不同。事实上这种差距是无关紧要的，这个理论中关键的一点就是，我们创造的任何计算模型都可以变形为与之等价的复杂模型。我们使用的方法是根据 20 世纪中叶麻省理工学院的马文·明斯基开发的一个模型改编的。

**图灵机模型** 我们研究的目标是一个抽象的机器模型，它比前一部分所提到的 DFA 模型要稍微复杂一些，这个模型被称为图灵机 (Turing Machine, TM)。在下面的描述中，我们会将它与图灵机的区别用蓝色明显地标识出来。每个 TM 由以下部分组成：

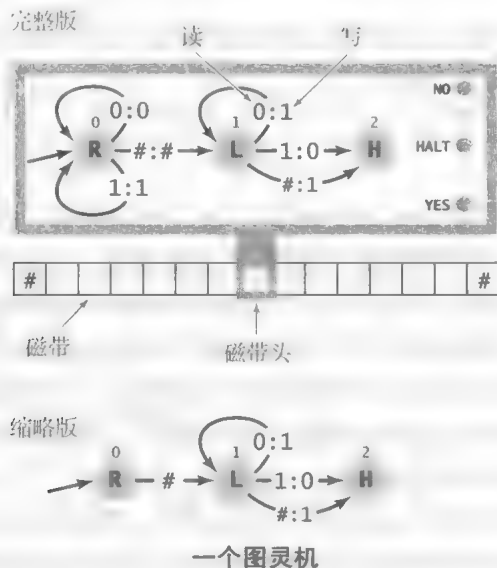
- 有限数量的状态，每个状态都被设定成左状态、右状态、暂停状态、接受状态和拒绝状态中的一个。
  - 一组用于指定下一个状态以及要写入哪个符号的迁移。每个状态对于符号表中的每个符号都有一个迁移。
  - 一个存有符号字符串的纸带。
  - 一个能够读或写一个符号并且能够向左或向右移动来读取下一个符号的纸带头。
- 一个 TM 操作从状态 0 开始，而且此时纸带头的位置在输入符号的第一个（最左侧），

766 然后读取和写入纸带并根据以下规则按照步骤改变状态：

- 读一个符号。
- 寻找与这个符号以及当前状态相关的迁移。
- 写入由这个迁移指定的符号。
- 改变状态为这个迁移指定的状态。
- 如果这个新状态是一个右状态，将纸带头向右移一个位置。
- 如果这个新状态是一个左状态，将纸带头向左移一个位置。
- 如果新状态是暂停、接受或拒绝状态中的一个，操作停止。
- 将对应于最终状态的灯点亮。

我们将每个 TM 表示为一个图，图中接受、拒绝、暂停、左和右状态分别是标记为 Yes、No、H、L 和 R 的顶点，每个迁移是一条被符号表中一对符号标记的有向边。

下图展示了一个基于二进制符号表的 TM。与 DFA 一样，我们从 0 开始用整数对顶点进行编号。每个迁移都标有一对用冒号分隔的符号。第一个符号标记的是迁移时读取的符号；第二个符号标记迁移时写入的符号。



为了减少混乱，我们通常使用一个简略的版本，在这个版本中我们没有画出盒子、指示灯还有纸带并且隐去了以下内容（即图中不再画出）：

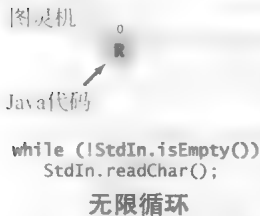
- 不会对纸带进行改变的自循环。
  - 如果一个状态迁移不会改变纸带，隐去其标签上的冒号和冒号后的第二个符号。
- 对任意的输入符号，如果它带来的状态转变没有更改纸带，我们将使用无标记的单箭头

来表示这个状态迁移。

我们将左右状态称为扫描状态（scanning state）——机器会扫描并掠过触发隐式转换的符号（读取然后写入相同的符号），直到读到要求转换到另一状态的符号。例如在上面描绘的机器中，状态 0 向右扫描直到遇到了输入中的一个 # 符号。缩略版可能会给你带来一些疑惑，但是它可以让机器变得更容易理解，在我们对它们进行更详细的研究时这个特征会变得更明显。

767

TM 模型和 DFA 模型之间的差异看起来很小，但是正如你将会看到的，它们具有很深远的意义。即使是仅仅只有几个状态的机器，它的行为也可能是非常复杂的。在我们开始之前，你有两个需要注意的地方。首先，纸带头何时向左右移动是没有限制的。这意味着纸带是无限的——往纸带的左侧或右侧读取或写入多少个纸带单元都是没有限制的。我们使用 # 这个元符号来表示那些尚未达到的纸带单元的内容，并且想象成这个纸带可以提供取之不尽的纸带单元，所以无论何时机器到达一个新的地方，都有一个符号为 # 的单元存在。其次，图灵机的状态迁移次数是没有限制的。这两种情况有一个最简单的例子，存在这么一个图灵机，它只包含一个右状态，且这个状态只有一个隐式转换，如左图所示。这台机器简单地向右移动，无限的要求读取越来越多的纸带单元。最开始这看起来像是一个人造的例子，但是它和一个对标准输入进行循环读取的 Java 程序几乎没有区别，因为我们都是假设 Java 程序可以从标准输入读取无穷无尽的数据。



接下来我们会对一些图灵机进行实验，这些图灵机可以完成有趣且有用的计算任务。

二进制增量。那么我们要介绍的第一个图灵机的例子进行了什么样的计算呢？这是一个二进制增量计算：如果输入是一个二进制数字，那么这个数字会增加。如果你对二进制的算术运算不熟悉的话，你可以现在就阅读 6.1 节，尽管这个计算很简单且很熟悉，你可能不需要这么做。考虑将二进制数 10111（十进制为 23）加 1 然后得到结果 11000（十进制为 24）的过程。其实这个计算的原理你已经学过，只是你在学校学到的是 999 加上 1 之后会得到 1000。为了简单解释这个过程，你可以这么说：从右向左数，将遇到的 1 都变成 0 直到你遇到了一个 0，然后将这个 0 变为 1。正如下面我们展示的详细跟踪一样，我们的 TM 实现了这个过程。从状态 0 开始，机器在所有的输入符号上向右扫描直到它遇到了输入的右侧的第一个 #，此时它开始向左移动并进入状态 1。处于状态 1 时，只要读数为 1，则向左扫描并将每个 1 改为 0。当它到达一个 0 时，将 0 转换为 1 并进行转换至暂停状态。在这个从右至左的扫描中，如果在遇到 0 之前先遇到一个 #，那么输入就全部是 1 并且已经全部变成 0 了，所以机器需要改变停止扫描处的 # 为 1（例如，#1111# 的增加结果是 #10000#——因为输入是无限的，所以仍然有一个 # 在开头处）。我们的例子也说明了即使输入拥有前导 0 时，机器也能正常工作。

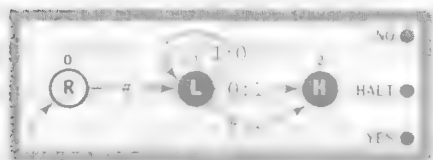
768

紧凑的跟踪格式。与其将整个 TM 的每一步都画出来，我们这里使用一种更为紧凑的跟踪格式来展示上面的过程。我们对当前状态、纸带头和纸带内容进行跟踪，将要写在纸带上的符号标成蓝色（图中的颜色和正文文字要对应）。在每一行中，我们都会标识出在每一个状态下纸带内容和纸带头位置发生的变化，并对应状态变化移动到新的一行。在每一行上，纸带头都会停在导致状态变化的符号的位置（这个符号可能会被改写）。

向右扫描#



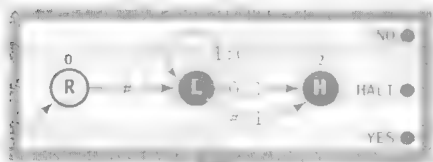
. . . # # # 0 1 1 # # # . . .



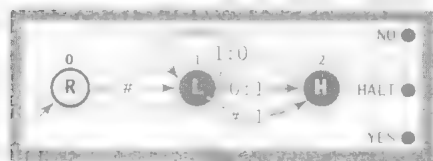
. . . # # # 0 1 1 # # # . . .



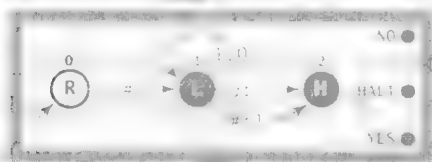
. . . # # # 0 1 1 # # # . . .



. . . # # # 0 1 1 # # # . . .

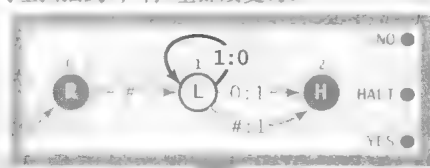


. . . # # # 0 1 1 # # # . . .

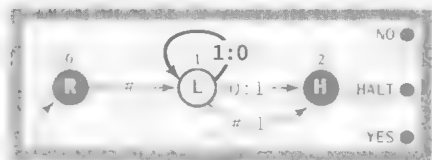


. . . # # # 0 1 1 # # # . . .

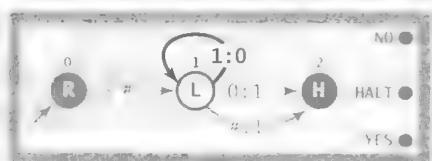
向左扫描到0, 将1全部改变为0



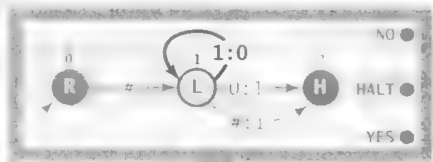
. . . # # # 0 1 1 # # # . . .



. . . # # # 0 1 1 # # # . . .

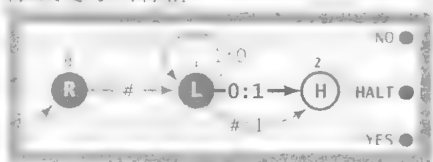


. . . # # # 0 1 1 # # # . . .



. . . # # # 0 1 1 # # # . . .

将0改变为1并停止



. . . # # # 0 1 1 # # # . . .



| 状态 | 磁带            | 说明            |
|----|---------------|---------------|
|    | # 0 0 1 1 1 # | 开始            |
| 0  | # 0 0 1 1 1 # | 向右扫描到#        |
| 1  | # 0 1 0 0 0 # | 向左扫描到0, 翻转每一位 |
| 2  | # 0 1 0 0 0 # | 停止            |

一个二进制增量 TM 的紧凑跟踪

相关机器。我们的二进制增量 TM 有一个显著的特点是它可以用于任意长度的整数。如果你的纸带上有一个十亿位的二进制整数，它也会增加它（如果十亿位都是 1，那么它会令十亿零一位变成 1）。实际上，我们可以用这么简单的机器来计算这么大的数字是一件非常了不起的事情。另外，如果我们将进入停止状态的 # 迁移的标签从 # : 1 改成 # : #，将得到一个固定长度的二进制增量器，它不会改变数字的长度，但是它会溢出忽略（就像许多计算机做的一样）。另一个变化思路是一个固定长度的二进制递减器（fixed-length binary decrementer），除了将机器中 0 和 1 的角色互换之外，这两个机器是相同的：从右向左移动，将遇到的 0 改变成 1 直到遇到一个 1，然后将 1 改变为 0。这个规则在除了这个数字全部是 0 的情况下是有效的。当数字全是 0 时，数字将全部变成 1（一个二进制递减函数是一个必须去除前导 0 的递增函数的反函数，见练习 5.2.10）。请注意，这三台图灵机除了离开状态 1 的迁移有一点区别之外，其他都是一样的。

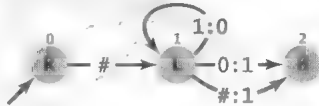
你需要通过练习写出不同输入下的机器跟踪轨迹，以测试你对 TM 的基本特性以及跟踪格式的理解。这会是一个对你很有帮助的过程。

状态表格表示。请特别注意，图灵机和 DFA 一样，我们可以很容易地将一个图灵机描述为一张表。表中的一行代表一个状态，对于每个输入符号，这一行显示这个状态是右、左、接受、拒绝还是暂停，并给出下一个状态和将要改写的符号。对于我们上面介绍过的二进制增量器 TM，我们在右侧给出了它的一个表格表示。当然，隐式状态迁移也会在这个表中明确地表示出来。

二进制加法器。下图展示的 TM 是一个加法器：它将纸带中输入的  $a+b$  替换为  $a$  和  $b$  的总和（ $a$ 、 $b$  和它们的和都是用二进制表示的正整数）。例如，如果机器启动时纸带里的内容是“#1011+1010#”，它会计算  $11_{10} + 10_{10} = 21_{10}$  并且在机器停止时将“#10101#”写在纸带上。与我们的二进制增量器相同，数字的大小是任意的——无论输入的数字有多少位，TM 都会计算它的总和。完成计算的策略如下，递增  $a$  的同时递减  $b$ ，循环操作直到  $b$  变成 0。机器通过 6 个状态来完成工作：

- 状态 0 直接向右扫描数据至最右端。
- 状态 1 对加号右侧的数字进行递减操作。
- 状态 2 向左扫描至加号左侧数字的最右端。
- 状态 3 对加号左侧的数字进行递增操作。
- 递减到 0 时，机器会达到状态 4——递减操作将所有的 0 都变成了 1，并且在查找 0 的时候找到了“+”。此时“+”左边的数字是“ $a+b$ ”，所以剩下需要做的事就是将“+”以及“+”右边的 1 全部变成 #。
- 状态 5 是停止状态。

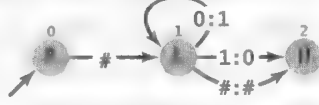
二进制递增器



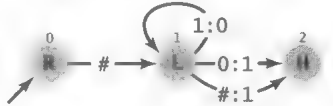
固定长度的二进制递增器



固定长度的二进制递减器



三个相关的 TM

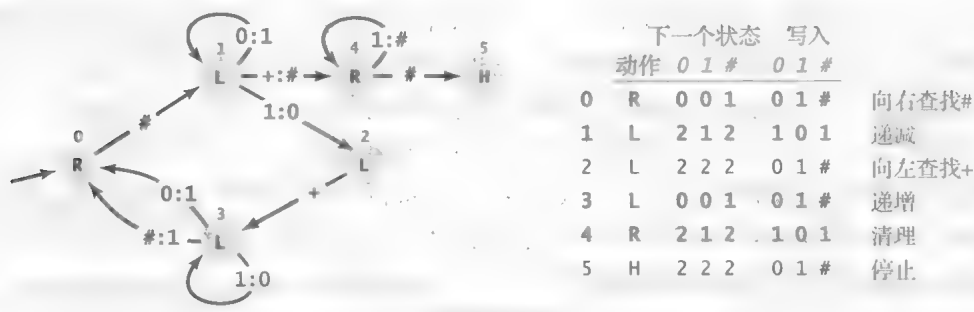


下一个状态 写入

| 状态 | 动作 | 0 | 1 | # | 0 | 1 | # |
|----|----|---|---|---|---|---|---|
| 0  | R  | 0 | 0 | 1 | 0 | 1 | # |
| 1  | L  | 2 | 1 | 2 | 1 | 0 | 1 |
| 2  | H  | 2 | 2 | 2 | 0 | 1 | # |

状态表格表示

769  
770



二进制加法器 TM

下面给出了一个跟踪的简单示例。经过对这个跟踪的学习，你会发现机器可以将任何两个给定的二进制数字相加。而且，这台机器可以计算任意长度的二进制数字。一个这么简单的机器可以执行这样的计算是很神奇的。

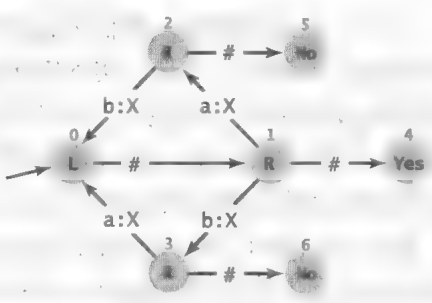
| 状态 | 磁带 |   |   |   |   |   |   |   |   |  | 说明     |
|----|----|---|---|---|---|---|---|---|---|--|--------|
| 0  | #  | 1 | 0 | 1 | + | 1 | 1 | # |   |  | 开始     |
| 0  | #  | 1 | 0 | 1 | + | 1 | 1 | # |   |  | 向右扫描至# |
| 1  | #  | 1 | 0 | 1 | + | 1 | 0 | # |   |  | 递减     |
| 2  | #  | 1 | 0 | 1 | + | 1 | 0 | # |   |  | 向左扫描至+ |
| 3  | #  | 1 | 1 | 0 | + | 1 | 0 | # |   |  | 递增     |
| 0  | #  | 1 | 1 | 0 | + | 1 | 0 | # |   |  | 向右扫描至# |
| 1  | #  | 1 | 1 | 0 | + | 0 | 1 | # |   |  | 递减     |
| 2  | #  | 1 | 1 | 0 | + | 0 | 1 | # |   |  | 向左扫描至+ |
| 3  | #  | 1 | 1 | 1 | + | 0 | 1 | # |   |  | 递减     |
| 0  | #  | 1 | 1 | 1 | + | 0 | 1 | # |   |  | 向右扫描至# |
| 1  | #  | 1 | 1 | 1 | + | 0 | 0 | # |   |  | 递减     |
| 2  | #  | 1 | 1 | 1 | + | 0 | 0 | # |   |  | 向左扫描至+ |
| 3  | #  | 1 | 0 | 0 | 0 | + | 0 | 0 | # |  | 递增     |
| 0  | #  | 1 | 0 | 0 | 0 | + | 0 | 0 | # |  | 向右扫描至# |
| 1  | #  | 1 | 0 | 0 | 0 | # | 1 | 1 | # |  | 递减     |
| 4  | #  | 1 | 0 | 0 | 0 | # | # | # | # |  | 清理     |
|    | #  | 1 | 0 | 0 | 0 | # |   |   |   |  | 停止     |

计算 5+3=8

效率。在本节和下一节中，我们把关注点仅放在探索那些可以用 TM 模型来实现的计算上；我们对执行这些计算的速度并不感兴趣，这是在 5.5 节重点讨论的问题。这样一来，你可能已经想知道一个这么慢的机器是否有存在的必要。在计算总和时，人们总是首先考虑字段长度，而不是它们的规格，我们甚至可以建立一个二次时间的 TM 来完成这个工作（见练习 5.2.22），只是这些与我们现在的讨论无关。

二进制字符中字符频率相等问题的求解。我们的下一个例子是一个已经被证明不可能由 DFA 实现的计算：给定一个二进制字符串作为输入，判断其中两个字符出现的次数是否相等。例如，对于 aabaabbbab 和 aaabbb 这样的字符串，我们的机器进入 Yes 状态，对于 aaa 和 aabbbab 这样的字符串，我们的机器进入 No 状态。这里展示的 TM 会执行这个计算。

|    |   | 下一个状态 |   | 写入 |   |   |   |         |
|----|---|-------|---|----|---|---|---|---------|
| 动作 | a | b     | # | a  | b | # |   |         |
| 0  | L | 0     | 0 | 1  | a | b | # | 向左查找#   |
| 1  | R | 2     | 3 | 4  | X | X | # | 测试第一个符号 |
| 2  | R | 2     | 0 | 5  | a | X | # | 向右查找b   |
| 3  | R | 0     | 3 | 6  | X | b | # | 向右查找a   |
| 4  | Y |       |   |    |   |   |   | 接受      |
| 5  | N |       |   |    |   |   |   | 拒绝      |
| 6  | N |       |   |    |   |   |   | 拒绝      |



判断二进制字符频率相等的 TM

772

如果你想要理解机器是如何工作的，请先研究下边给出的跟踪轨迹。它先找到最左边的符号（a 或 b），然后用 X 覆盖这个符号，接着查找另一个符号（b 或 a）。如果找不到匹配的字符，那么机器进入 No 状态。若找到了匹配的符号，机器会用 X 覆盖。然后机器接着返回输入的最左端并查找下一个匹配的字符对。每当机器进入状态 0 时，我们知道机器已经用 X 覆写了相等数量的 a 和 b，所以如果在状态 1 时扫描没有找到 a 或者 b 符号，那么证明原始字符串具有相同数量的 a 和 b，这个字符串被接受。与 Java 程序一样，我们没有提供完整的证明，所以如果你的研究方向偏向于数学，你可以想象一下如何为一个图灵机提供一个理论上的证明。

| 状态 | 磁带 |   |   |   |   |   |   |   |   |  | 说明      |
|----|----|---|---|---|---|---|---|---|---|--|---------|
|    | #  | a | a | b | b | b | a | b | # |  | 开始      |
| 0  | #  | a | a | b | b | b | a | b | # |  | 向左扫描至#  |
| 1  | #  | X | a | b | b | b | a | b | # |  | 测试第一个符号 |
| 2  | #  | X | a | X | b | b | a | b | # |  | 向右扫描至b  |
| 0  | #  | X | a | X | b | b | a | b | # |  | 向左扫描至#  |
| 1  | #  | X | X | X | b | b | a | b | # |  | 测试第一个符号 |
| 2  | #  | X | X | X | X | b | a | b | # |  | 向右扫描至b  |
| 0  | #  | X | X | X | X | b | a | b | # |  | 向左扫描至#  |
| 1  | #  | X | X | X | X | X | a | b | # |  | 测试第一个符号 |
| 3  | #  | X | X | X | X | X | X | b | # |  | 向右扫描至a  |
| 0  | #  | X | X | X | X | X | X | b | # |  | 向左扫描至#  |
| 1  | #  | X | X | X | X | X | X | X | # |  | 测试第一个符号 |
|    | #  | X | X | X | X | X | X | X | # |  | 在状态6时拒绝 |

等价判定轨迹

你可以在本节末尾的练习中找到许多其他类型的图灵机，在本章的后面我们也会遇到很多其他的例子。我们设计出的图灵机，可以用来进行各种计算，囊括了从乘法、除法，到一元 - 二元的换算，再到龙形曲线的函数式。设计一个图灵机就像编写一个 Java 程序，是一个有趣且令人满意的智力体验。我们之所以列举这么多例子和练习，目的是为了让你相信，任何你能想象到的计算都可以用一个图灵机来实现。这就是图灵在 1937 年的论文中的重大发现。

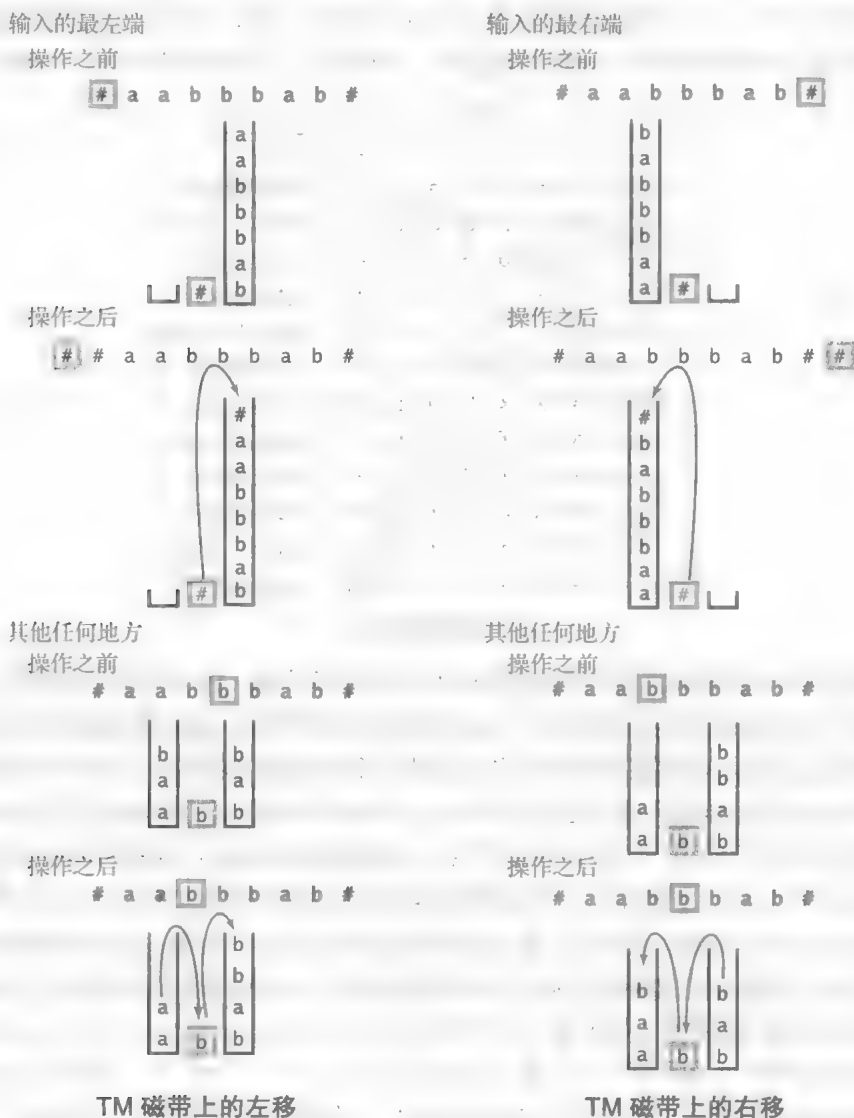
与 DFA 一样，如果你没有用于调试和跟踪的 TM 编程环境，那么你肯定无法深入了解我们练习中复杂 TM 的设计和开发。或许正如你预料的一样，创建一个可以模拟任何 TM 的 Java 程序并不是一件困难的事情。我们会使用这样一个 Java 程序来测试本书中的 TM，并产生对这些 TM 的跟踪轨迹。接下来，我们研究这个程序。

773

**通用虚拟图灵机** 在程序 5.1.3 中，我们给出了一个 Java 程序，你可以通过这个 Java 程序模拟各种 DFA 的操作，生成它们在各种输入下的轨迹来研究 DFA 的属性。现在我们来对图灵机做相同的事。



纸带。第一个挑战是纸带。我们如何模拟一个长度为无限大的纸带？你自己可能要花很长时间来思考这个问题，但是其实答案十分简单：将当前的字符保存在一个 `char` 型变量 `current` 中，然后使用两个下推栈，一个用于纸带头的左端（就好像字符在被从左往右推入栈），另一个用于纸带头的右端（就好像字符在被从右往左推入栈）。一旦建立起这个表示形式，我们就能很容易地看到解决这个问题的方法。例如下图所示，如果纸带头不在输入最左端的位置，我们可以通过某种方法将纸带头向左移动。这个方法是将变量 `current` 压入右堆栈，然后将左堆栈推出给变量 `current`。如果纸带头位于输入的最左端，我们只需要将一个 `#` 压入到右栈上，就如下图所示的左上方所示。向右移动的两个例子相似，如下图所示的右边所示。鉴于这种设计，我们在程序 5.2.1 中很简单地实现了它。这段代码是对于无限长度纸带的一个很好的表达：每当客户程序要求移动到目前为止所看到的所有数据的最左或最右端时，我们简单地制造一个 `#` 符号让客户程序产生纸带的长度是无限的幻觉。实际上纸带并不是无限长的，但是从客户程序的角度上来看，纸带的长度没有限制。当然，栈也具有相同的属性，所以它在模拟无限长度的纸带时十分有用。



机器。TuringMachine (程序 5.2.2) 是一个 Tape 类的客户程序, 它实现了一个通用虚拟 TM, 这个 TM 可以模仿任何 TM 的操作。与程序 5.1.3 中的 DFA 一样, 它从标准输入中提取一个 TM 的规范 (一个由命令行参数指定名称的文件) 和一个字符串序列, 然后以给定字符串运行 TM, 并将结果打印出来。

TM 的文件输入格式首先是状态的数量, 其次是字母表, 然后对于每个状态后都有一行信息。每一行中都包含一个字符串, 这个字符串是 L、R、H、Yes 和 No 其中的一个。每个 L 或者 R 后面都有一个与每个字母表中的符号对应的状态索引, 第  $i$  个状态索引给出了当纸带头包含的是字母表中的第  $i$  个符号时, 从该状态要如何进行迁移。对于字母表中的每个符号, 状态索引后面还有一个对应的符号。这个符号表示的是当纸带头包含的是字母表中的第  $i$  个符号时, 第  $i$  个符号将要写回进纸带的符号。程序 5.2.2 下面的文件 addTM.txt 定义了我们的加法器 TM。

774  
775

程序5.2.1 虚拟图灵机的磁带

```
public class Tape
{
 private Stack<Character> left = new Stack<Character>();
 private Stack<Character> right = new Stack<Character>();
 private char current;

 public Tape(String input)
 {
 right.push('#');
 for (int i = input.length() - 1; i >= 0; i--)
 right.push(input.charAt(i));
 current = right.pop();
 }

 public char read()
 { return current; }

 public void write(char symbol)
 { current = symbol; }

 public void moveLeft()
 {
 right.push(current);
 if (left.isEmpty()) left.push('#');
 current = left.pop();
 }

 public void moveRight()
 {
 left.push(current);
 if (right.isEmpty()) right.push('#');
 current = right.pop();
 }

 public String toString()
 { /* 见练习 5.2.7 */ }
}
```

这个程序使用两个下推栈来模拟图灵机需要的无限长磁带。磁带头下的符号保存在变量current中, 磁带头左侧的符号保存在左下推栈中, 磁带头右侧的符号保存在右下推栈中。

776

程序5.2.2 通用虚拟TM

```

public class TuringMachine
{
 private String[] action;
 private ST<Character, Integer>[] next;
 private ST<Character, Character>[] out;

 public TuringMachine(String filename)
 { /* 见文本 */ }

 public String simulate(String input)
 {
 Tape tape = new Tape(input);
 int state = 0;
 while (action[state].equals("L")
 || action[state].equals("R"))
 {
 if (action[state].equals("R")) tape.moveRight();
 if (action[state].equals("L")) tape.moveLeft();
 char c = tape.read();
 tape.write(out[state].get(c));
 state = next[state].get(c);
 }
 return action[state] + " " + tape;
 }

 public static void main(String[] args)
 {
 TuringMachine tm = new TuringMachine(args[0]);
 while (StdIn.hasNextLine())
 StdOut.println(tm.simulate(StdIn.readLine()));
 }
}

```

对于标准输入中的每个字符串的第一个命令行参数，这个程序模拟了TM的操作。如果TM停止了，程序将打印Yes、No或者Halt，然后是磁带的內容。否则，这个程序可能会陷入无限循环。

```

% more addTM.txt
6 01+#
R 0 0 0 1 0 1 + #
L 1 2 4 1 1 0 # #
L 2 2 3 2 0 1 + #
L 0 3 3 0 1 0 + 1
R 4 4 4 5 # # # #
Halt

```

```

% java TuringMachine addTM.txt
101+11
Halt 1000
10000000011011+11000001
Halt 10000011011100

```

构造函数中需要创建必要的数据结构，这些数据结构应根据给定的命令行参数文件展现TM的内部信息，如下：

- 读取字母表和状态数量。
- 为状态动作创建一个字符串数组，为L和R两个状态创建两个符号表数组，其中一个用于存放状态迁移，另一个用于存放写入的符号。
- 通过读取给定文件的每个状态的信息来填充这些数据结构。

实现构造函数的代码很简单：

```

public TuringMachine(String filename)
{
 In in = new In(filename);
 int n = in.readInt();
 String alphabet = in.readString();
 action = new String[n];
 next = (ST<Character, Integer>[]) new ST[n];
 out = (ST<Character, Character>[]) new ST[n];
}

```

```

for (int st = 0; st < n; st++)
{
 action[st] = in.readString();
 if (action[st].equals("Halt")) continue;
 if (action[st].equals("Yes")) continue;
 if (action[st].equals("No")) continue;

 next[st] = new ST<Character, Integer>();
 for (int i = 0; i < alphabet.length(); i++)
 {
 int state = in.readInt();
 next[st].put(alphabet.charAt(i), state);
 }

 out[st] = new ST<Character, Character>();
 for (int i = 0; i < alphabet.length(); i++)
 {
 char symbol = fn.readString().charAt(0);
 out[st].put(alphabet.charAt(i), symbol);
 }
}
}
}

```

778

程序 5.1.3 中的其他方法也很简单。`simulate()` 方法模拟了 TM 的操作，而 `TuringMachine` 的 `main()` 方法在标准输入的每一行上都调用了 `simulate()` 方法。

程序 5.1.3 既是一个 TM 由什么构成的完整规范，也是研究特定 TM 属性的一个必不可少的工具。只要你提供字母表、一个 TM 的表格描述和一系列输入字符串，程序就会模拟 TM 执行对应每个字符串的操作。这个简单的例子向我们解释了虚拟机的概念。这不是一个真正的计算设备，而是对这类设备将如何工作的一个完整定义。你可以把 `TuringMachine` 想象成一个“计算机”，你可以通过指定一组遵循合法 TM 规则的状态和迁移在这台“计算机”上进行“编程”。每个 TM 是这台计算机上的一个“程序”。

我们在描述图灵机的时候使用的词、与在 5.1 节中描述如何实现一个通用虚拟 DFA 中用到的十分相像，但是 DFA 和图灵机之间存在深刻的区别：一个图灵机可能不会停止。如果给定的图灵机在给定的输入上会进入无限循环，那么程序 `TuringMachine` 也会如此。如何防止这种情况发生，你会在 5.4 节中学到。

图灵机的模型非常简单。它只是一个玩具吗？当然不是！在第 6 章和第 7 章中，你可能会惊讶地发现，你正在使用的计算机实际上也是基于一个计算模型建立起来的，这个计算模型更接近于图灵机模型，而不是你熟悉的 Java 环境。更重要的是，图灵机模型让我们能解决一系列关于计算性质的深刻问题，接下来我们会研究这个问题。

779

## 问答环节

**问：**我可以从哪里了解更多关于图灵机的知识？

**答：**许多书籍都涉及这个话题。我们在文中提到了马文·明斯基的《计算：有限与无限机器》。如果你想知道这方面更先进的知识，你可以看看迈克尔·西普塞的《计算理论导引》或者大卫·哈雷尔写的《计算机的极限：它们真正不能办到的事》。

**问：**我可以从哪里了解更多关于艾伦·图灵的知识？

**答：**有几本值得注意的传记记载了艾伦·图灵的生活故事和遗留财富，这之中就包括安德鲁·霍奇斯写的《艾伦·图灵传》。由本尼迪克特·康伯巴奇主演的《模仿游戏》就是基于霍奇斯写的这本传记拍摄的一部电影。



机生成的轨迹，这个轨迹就如练习 5.2.8 中所示。

- 5.2.9 设计一个图灵机，用于识别一种语言的字符串是否包含相同数目的符号 A、B 和 C。
- 5.2.10 设计一个二进制减法器图灵机，这个图灵机与我们的二进制加法器正好相反，同时它能删除所有前导 0。例如，#1000# 递减的结果应该为 #111#<sup>⊖</sup>。
- 5.2.11 设计一个判定某种语言的图灵机，这种语言所有符号的数量为 2 的幂次方。
- 5.2.12 设计一个判定某种语言的图灵机，这种语言由长度为奇数且最中间的符号为“|”的二进制字符串组成。
- 5.2.13 设计一个判定某种语言的图灵机，这种语言的字符串形式为：前面为  $n$  个 a 符号后面跟  $n$  个 b 符号，其中  $n$  为正整数。
- 5.2.14 设计一个判定某种语言的图灵机，这种语言由所有二进制回文组成。
- 5.2.15 设计一个判定某种语言的图灵机，这种语言由所有包含完整格式的圆括号的字符串组成：(), (()), ((())), (((()()))) 等。
- 5.2.16 设计一个判定某种语言的图灵机，这种语言为：在两个相同的十进制数字间插入一个“|”符号隔开。
- 5.2.17 设计一个可以为输入的纸带制作副本的图灵机。例如纸带一开始记录的是 abcd，则最终纸带上应该留下 abcd#abcd。
- 5.2.18 设计一个图灵机，这个图灵机会收到由“x”分隔的两个一元字符串，将这两个字符串相乘，然后在纸带的两个字符串后面加上等号，再在等号后面加上结果。例如输入 11x11111，纸带上的结果应该为：11x11111=111111111。
- 5.2.19 设计一个用二进制计数的图灵机。最开始时纸带应该是空的。随机机器的运行，纸带应该包含 1、10、11、100、101、110 等，并且不停地继续增加。
- 5.2.20 设计一个图灵机，这个图灵机会收到由“^”分隔的两个相等长度的二进制字符串，并且在纸带上留下两个字符串按位取或的结果。
- 5.2.21 设计一个图灵机，它以最前位为 1 的二进制整数为输入，并且在纸带中留下这个整数的位数的二进制表示。这个函数被称为离散二进制对数函数。

782

783

## 创新练习

- 5.2.22 有效的加法器。设计一个与正文中所提到的图灵机类似的图灵机，这个图灵机以两个用“+”分隔的二进制字符串为输入，将这两个字符串解释成二进制数字，并将它们相加，然后将结果留在纸带上。不同于文中的 TM，你的机器的运行时间应该由一个关于数字长度的多项式决定，而不是这些数字的大小。
- 5.2.23 有效的比较器。设计一个与文中所提到的图灵机类似的图灵机，这个图灵机以两个用“?”分隔的二进制字符串为输入，将这两个字符串解释成二进制数字，如果第一个数字小于第二个则进入“Yes”状态，否则进入“No”状态。确保你的机器的运行时间应该由一个关于数字长度的多项式决定，而不是这些数字的大小。
- 5.2.24 龙形曲线。设计一个图灵机，这个图灵机的输入纸带最初只有一个有  $2^n-1$  个 0 的序列，最后会在纸带上留下指令来绘制一条龙的曲线（见练习 1.2.35）。使用以下算法：在 L 和 R 之间交替地使用 L 或 R 来替换 0。

⊖ 此处原书中有错误。——译者注

5.2.25 克拉茨 TM。设计一个图灵机，这个图灵机以一个整数的二进制表示作为输入，如果这个整数是偶数，则对它除以 2；如果这个数是奇数，则对它乘以 3 后加 1，重复此过程直到结果等于 1。这就是克拉茨函数，关于这个机器是否对于所有的输入都能终止，这个问题仍然在讨论当中（见练习 2.3.29）。

784  
785

## 5.3 普遍性

图灵机模型本质上是一个数学作品，它将有关计算的基本数学概念简化，以使我们可以得出精确结论。这项工作的基础是由图灵在他最初的论文里提出来的，但随后学者们不断地发展，直到今天我们被各种计算设施所包围，都有赖于图灵机模型得出的这些结论。图灵机有如此巨大的影响力，我们所有从事于计算的人（当然，也包括本书的读者）都有必要对其有个基本理解。在本节中，我们试图讲解图灵在那篇论文中提出的两个基本概念：一是一台通用目的的计算机可以进行任何计算；二是假设所有计算设备本质上是等价的。

**算法** 当我们写一个计算机程序时，我们通常是在设计一种可以解决某个问题的方法。这个方法必须依赖于程序环境，或者这个方法适用于很多环境。正是方法，而不是程序本身，带领我们一步步地解决问题。在计算机理论中，名词“算法”（algorithm）用来描述实现计算机程序中有限的、确定的、有效地解决问题的办法。算法是计算机科学理论中的核心领域。

在本书中，我们已经学习了很多算法，比如牛顿定理、欧几里得定理、归并排序、广度优先搜索等。这里给出的定义虽然不够严谨，但也足够说明这个概念。图灵机给数学家们带来了这个概念，并可用于数学证明。在这里，我们关注于这些重要的思路，而不再停留于定义及其完整版证明。

786

**可判定性。**假设我们在纸带上输入一个字符串并将一个图灵机处于启动状态，有四种可能的结果。该机器可能：

- 停在标有“YES”的状态（接受输入字符串）。
- 停在标记为“NO”的状态（拒绝输入字符串）。
- 停在标有“H”的状态（停止，不再接受或拒绝）。
- 以上都不是（进入无限循环）。

目前，当机器停机时（或处于无限循环中），我们会忽略纸带前后的内容。我们说一个图灵机识别某个语言，是指图灵机以这个语言的所有可能的字符串作为输入时，都会引导机器最终走到接受状态。同时，对于所有不在所识别的语言中的输入字符串，图灵机都会停止运行（终止于标记为“NO”或“H”的状态），那么我们就说这个图灵机可以判定（decide）该语言。对于每一个输入字符串，这个图灵机必须都能停机并且给出正确的答案。例如，我们在前一节末尾设计的二进制频率计数器图灵机能够判定（和识别）所有 a 和 b 数量相同的二进制字符串。注意：对于 DFA，当 DFA 总是停止时，我们不需要区分判定和识别。

**可计算性。**把输出纸带也考虑在内，我们就可以将图灵机的可用性和可判定性扩展到函数的属性。例如，练习 5.2.21 中要求 TM 计算离散二进制对数函数。如果在某个图灵机上，当它的纸带被初始化为  $x$ ，它就将  $f(x)$  的计算结果留在纸带上，我们就说函数  $f(x)$  是可计算的（computable）。我们可以将整数表示为二进制字符串或十进制数字的字符串，或使用任何其他合理的表示。所有关于整数的常用操作（增量、加法、减法、乘法、模数除法、指数）都是可计算的函数。



这些可判定性和可计算性的概念帮助我们精确地捕捉了术语“算法”的含义：判定某种语言或计算某个函数的图灵机就代表了完成该任务的算法。正如我们经常考虑多种算法来解决编程问题一样，每个特定任务可能会有许多图灵机。在计算理论中，图灵机和算法这两个术语可以互换使用，使我们能够用严谨的数学方法对“所有算法”或“针对特定任务的所有算法”进行详尽的陈述。

787

**处理程序的程序** 为了让你进入理论讨论的思维模式，我们先离题，简要地讨论一下“处理程序的程序”这一概念，即程序将另一个程序作为输入（并且可能产生另一个程序作为输出）。经过回顾，你一定会意识到经常用到处处理程序的程序。例如，移动设备上的每个应用程序都是一个程序：当你下载一个新的应用程序时，你会用到服务器主机上的一个程序，这个程序使用你要下载的 App 程序作为输入，并将它打包输出（发送给你）；同样的道理，在移动设备上存在另一个程序，该程序以刚才的 App 程序作为输入（以接收应用程序），也会有另一个程序（操作系统）来启动这个 App 程序。

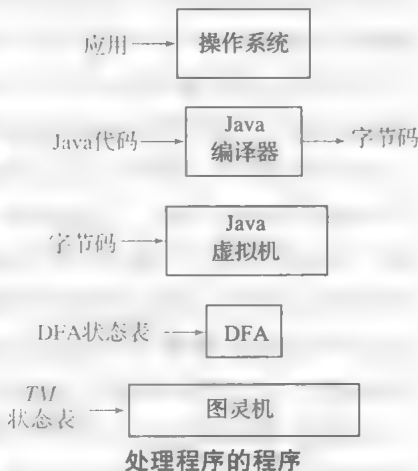
Java 开发环境。我们在 1.1 节中介绍过 Java 开发环境中两个处理程序的程序，就是我们一直用到现在的 Java 编译器（我们将在 6.4 节中再详细介绍其原理）。Java 编译器将 Java 代码转换为另一种称为 Java 字节码的编程语言，然后由（通用）Java 虚拟机（JVM）将以该字节码编写的任何程序作为输入，并最终使用另一种编程语言（即计算机的机器语言）来运行它。JVM 实际上是用另一种编程语言，即 C 语言编程语言编写的程序，C 语言比 Java 更老、更底层，但也常见于大多数计算机。

通用虚拟 DFA（Java 实现）。我们的程序 DFA（程序 5.1.3）以 DFA 的状态表描述作为输入，然后模拟该机器的操作。正如我们在 5.1 节结束时所讨论的那样，可以这样理解我们的工作，将状态表的编写规则作为编程语言，将每个 DFA 状态表描述的定义作为以该语言编写的程序。所以 DFA 是一个 Java 程序，它将“程序”（特定 DFA 的描述）作为输入。另外，执行 5.1 节末尾描述的将任何 NFA 转换为 DFA 的过程（请参见练习 5.1.39），需要一个 Java 程序将一个程序（一个 NFA 转换表）作为输入，并生成一个程序（一个 DFA 状态表）作为输出。

通用虚拟 DFA（TM 实现）。当然，并不是每个程序都需要在 Java 中实现！事实上，甚至有可能创建一个处理程序的图灵机。例如，练习 5.3.2 中描述的 TM 模拟了一个使用二进制符号表的三态 DFA 的操作（采用相同的设计进行直接的扩展，可以处理更多状态和更大符号集）。它需要（在 TM 纸带上）输入任何三态 DFA 的状态表和该 DFA 的二进制输入字符串，并根据该输入模拟 DFA 的操作，如果输入字符串是可以被 DFA 接受的语言，输出状态码为 Yes，如果不是则为 No（见练习 5.3.4）。在这种情况下，我们有一个 Java 程序，它把一个程序（一个 TM 状态表）作为输入，这个程序就是以程序（一个 DFA 状态表）作为输入！

总之，处理程序的程序是计算的基础，我们当然可以创建处理程序的图灵机。图灵在计算机出现之前就构想了这些事实的意义，这是非常了不起的。

通用图灵机。由于我们可以创建一个 TM 来模拟任何给定输入上的任何 DFA 的操作，那么我们可以创建一个 TM 来模拟任何给定输入上的任何图灵机的操作吗？这个问题的答案



788

是肯定的！这是图灵论文的贡献。这种机器被称为通用图灵机（Universal Turing Machine, UTM）。

我们在本书中没有开发完整的 UTM，但是如果你尝试完成练习 5.3.2，就可以看到如何解决这个问题：

- 扩展输入格式以包含重写符号。
- 开发一个 TM 的附属程序 Tape（程序 5.2.2），以模拟正被模拟的机器的纸带。
- 将纸带操作添加到练习 5.3.2 实现的模拟通用虚拟 DFA 的 TM 中。
- 添加一个机制来跟踪当前状态（而不是复制所有状态的“代码”，这需要知道状态的数量）。

在目前的情况下，我们并不是要求你实现并确认所有这些细节（即使图灵最初的 UTM 也有漏洞），只是想说服你，自己建造这样一台机器是一个合理的练习，虽然这已经超出了本书的范围。现在你已经知道 UTM 是存在的，那么你就已经为接下来我们要研究的实现细节做好了准备。

789

如果你有兴趣研究细节，你可以在我们的网站上找到一个 24 个状态、7 个符号的 UTM，以及一个图形化的虚拟通用 TM，你可以利用它来直观地追踪操作过程。

通用计算机。假如你的计算机上安装有 Java，并能够运行 TuringMachine，那它就是一个通用的虚拟图灵机：它能够运行不同的算法，而不需要任何硬件修改。你将在第 6 章和第 7 章中看到细节，但是现在简单地意识到这个任务的可行性就好了。因为现代处理器基于冯·诺依曼架构，在这种架构下计算机程序和数据都存储在主存储器中。这就意味着，内存的内容是被视为机器指令还是数据，取决于程序执行的上下文。这种安排与 UTM 的纸带完全相似，它也是由一个程序（原始 TM）及其数据（原始 TM 的纸带内容）组成。

因此，图灵在 UTM 方面的工作预见了通用计算机的发展，从法律的角度甚至可以把他们视为软件的发明人！你可以设计一台机器并为它编程来执行各种任务，而不是为不同的任务设计不同的机器。例如，你可以用同样的设备来分析实验数据、撰写论文；处理图片、音乐和电影；浏览网页；通过社交媒体进行沟通以及下棋等。

790

**邱奇 - 图灵论题** 图灵坚信，他的模型体现的理念是一个理想化的数学家按照明确的程序进行计算的过程。几乎在同一时间（实际上早一点），阿隆索·邱奇（Alonso Church）发表了一个完全不同的数学模型来研究可计算性的概念。事实证明，邱奇的模型，即 Lambda 演算（lambda calculus），直接导致了现代函数式编程语言的发展。在计算方面，这些模型似乎有很大不同，但图灵后来证明它们是等价的，因为它们都表征了完全相同的一组数学函数。虽然他们的研究纯粹是理论性的，但这个等价的结论使得图灵和邱奇在计算的研究上得出了一致的结论。图灵关注的是一个数学家理想化地对整数函数的计算，邱奇关注的是纯函数，但最终他们意识到他们对于真实世界的抽象有一点是相同的：所有物理上可实现的计算设备都可以被图灵机模拟。这就将现实世界中计算的研究简化到对图灵机的研究，有效地控制了计算设备的类型。我们用通用图灵机的概念来正式地表述这个论题：

**邱奇 - 图灵论题** 通用图灵机（UTM）可以执行任何由物理上可实现的计算设备描述的（如判定语言或者完成计算函数）。

说明：在图灵和邱奇构想的形式中，这个论题是关于一个“理想化的数学家”按照一个明确的程序进行计算的想法。在这里，论题表述的是在我们的宇宙中，关于自然规

律和哪些计算可以被完成的声明。

邱奇 - 图灵论题不是一个数学表述，也不是可以严格证明的论题。这是因为我们不能在数学上定义“物理上可实现的计算设备”的含义。

自第一次制定到现在多年以来，正如我们看到的，已经有大量的证据支持这个论点。但是，这个论题可能会被驳斥（被证明是错误的）。如果有人发现一个更强大的物理可计算的计算模型，我们将不得不放弃或修改它。

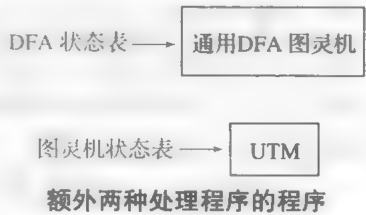
这个论题的逆否命题也是有趣的。它表明，如果一个计算任务不能在图灵机上完成，那么使用任何物理上可实现的计算设备也是无法完成的。正如我们将在 5.4 节看到的那样，确实有图灵机不能识别的语言。因此，如果我们相信邱奇 - 图灵论题，我们就必须承认，我们在物理上可实现的计算设备能够完成的计算任务是有限制的。

**TM 模型的变体** 一些支持邱奇 - 图灵论题的证据与研究图灵机模型的改变是否有作用有关。一方面，对于某种 TM 无法识别的语言或者无法计算的函数，研究人员一直在研究能否向机器中添加机制来使得可以识别或计算；另一方面，研究人员一直在研究如何在不削弱模型的功能前提下简化图灵机模型。请记住，在目前的情况下，我们并不关心一台机器可能使用多少个状态或符号，或者一个计算可能需要多少个状态迁移——而是只关心计算能力，即多少语言可以被识别、多少函数可以被计算。

791

**等效模型。**本书作者习惯于使用图灵原始模型的轻微变体，就像我们前文讨论过的 Minsky 开发的模型，通常非常容易观察出来发生了什么改变，因此不再进行额外的说明。其实这些选择只是为了方便起见。我们相信，我们的版本及其生成的表格简单易懂。关于计算理论的高级教科书的作者可能会选择不同的版本，从而生成更简单的证明。例如，常见的选择是将纸带头移动与每个状态迁移相关联，而不是与目标状态相关联。我们在 5.2 节中提到的另一个等效模型是一个双栈下推自动机——这个名字非常直观，我们在程序 TuringMachine 中确实使用两个栈。

**增强。**关于 TM 模型的研究已经非常深入，并且得到了一系列可能的改进。例如，是否会添加另一个带有独立纸带头的纸带使 TM 模型更强大？这个问题的答案是否定的，因为我们可以用 TM 来模拟这样的机器，用纸带奇数位置表示一个纸带，偶数位置表示另一个纸带。再举一个例子，考虑为系统添加非确定性功能的价值。结果是这不会提升 TM 的功能，因为使用类似于为 NFA 构造 DFA 的方法，我们可以为不确定性 TM 构建一个确定性 TM，它识别相同的语言或者计算功能可以完全相同。下面会有更多的示例。我们省略了进一步的讨论和证明，要指出的是，尽管经过了数十年的努力，仍没有人成功找到更强大的、物理可实现的模型。



**限制。**研究人员还试图简化图灵机模型。在下面的表格中显示了几个例子。例如，仅顺着一个方向使用无限纸带是没有限制的，因为我们可以在一个方向上使用奇数位置，另一个方向使用偶数位置。作为另一个例子，使用二进制符号表是没有限制的，因为我们可以用二进制编码纸带符号。我们再次省略细节和进一步的讨论证明。当然，正如我们所看到的，DFA 模型不如 TM 模型功能强大，DFA 可以看作一个不能写入纸带且纸带只能在一个方向上移动的 TM。寻找最简单的 TM 模型是吸引了许多研究人员的目标，我们将很快回到这个话题。

792

| 图灵机变体 | 描述                          |
|-------|-----------------------------|
| 等效模型  |                             |
| 移动迁移  | L和R移动与每个迁移相关联，而不是目标状态       |
| 双栈PDA | 双栈下推自动机                     |
| 增强    |                             |
| 多带    | 添加有限数量的带<br>独立磁带头的独立磁带      |
| 多维    | 使用多维磁带（允许磁带头向任何方向移动）        |
| 不确定性  | 允许任何输入字符的多重迁移               |
| 基于概率  | 迁移是随机选择的（如果大多数结果导致接受状态，则接受） |
| 不经意性  | 迁移不依赖于输入                    |
| 限制    |                             |
| 单向无限  | 磁带只有一个方向是无限的                |
| 二进制   | 只允许两个符号                     |
| 双态    | 只允许两种状态                     |
| 非擦除   | 没有覆盖能力                      |
| 顺序    | 状态是环形排列的（只能从一个状态转换到下一个状态）   |

#### 功能没有受到影响的 TM 变体

对于同一种语言的识别问题或者同一个计算函数，有数以百计甚至数千篇的论文和书籍来讨论图灵模型的变体。这一事实无疑表明，该模型至少是我们理解计算的一个转折点——从单栈 PDA 到图灵等效双栈 PDA 的步骤确实是一个巨大的进步。

**通用模型** 如果模型等同于图灵机模型（它可以识别同一组语言或完成相同的计算函数），则称模型为图灵完备模型或图灵通用模型。邱奇-图灵论题表明，图灵机是自然界中的一个基本对象。是否有其他计算模型可以像图灵机那样运行基于任何输入的任何程序呢？当然，答案是肯定的！一个世纪以来，许多数学家、计算机科学家、物理学家、语言学家和生物学家已经考虑了许多其他的计算模型，这些模型已被证明是图灵完备模型。在后面的表格中列出了其中的一部分。我们在这里只重点介绍其中的一些内容。

**拉姆达演算 (Lambda calculus)**。正如前文提到过的，当图灵正在普林斯顿大学准备他的论文时，邱奇正在完成他的拉姆达 ( $\lambda$ ) 演算工作，这是一个正式的系统，是现代函数式编程的基础。认识到邱奇的拉姆达演算和图灵的机器模型是相同的，这也正是邱奇-图灵论题的论点。

**计数机**。比图灵机更简单的模型是计数机，它由明斯基推广。其中纸带由一小组计数器代替，这些计数器可以保存任何整数；由一小部分指令取代一组操作来表示状态和迁移，如“增量”“减量”和“如果为零则跳跃”。

**元胞自动机**。你可能熟悉生命的游戏 (Game of Life)，这是由数学家约翰·康威设计的计算模型（详见练习 2.4.20 和练习 5.3.19）。这个游戏是元胞自动机的一个例子，元胞自动机是细胞与邻居交互的离散系统。元胞自动机的研究始于 20 世纪 40 年代，其提出者约

翰·冯·诺依曼是计算史上的一个重要人物，你将在第 6 章看到。

794

你的计算机。正如我们已经知道的，TuringMachine（程序 5.2.2）证明你的计算机至少与任何图灵机一样强大，因为它可以模拟任何图灵机的行为。但是，你可能会惊讶地发现存在可以模拟你的计算机操作的图灵机。正如将在第 6 章和第 7 章里介绍的，你会发现计算机的基础是一个包含处理二进制数字指令的机器模型。相比 Java 环境，这个机器模型与图灵机更类似。开发能够模拟这种机器的图灵机在概念上并不困难。因此，如果你可以在计算机上开发可以判定某种语言或计算某些功能的程序，那么就存在可以执行相同任务的图灵机（并且任何通用图灵机也可以执行此操作）。

编程语言。几乎目前使用的所有编程语言都是图灵完备模型，包括过程编程语言（如 C、Fortran 和 Basic）、面向对象的编程语言（如 Java 和 Smalltalk）、函数式编程语言（如 Lisp 和 Haskell）、多范式语言（如 C ++ 和 Python）、专用语言（如 Matlab 和 R）和逻辑编程语言（如 Prolog）。虽然有些编程语言似乎比其他编程语言更强大，但它们的核心都是等效的（就它们可以实现哪些计算函数以及它们可以判定哪些语言而言，是等效的）。编程语言在其他重要特性（如可维护性、可移植性、可用性、可靠性和效率）方面会有所不同，因此我们选择语言时是基于方便性和效率，而非能力。

字符串替换系统。许多模型涉及创建一组规则来替换字符串集中的子字符串。这样的系统可以非常简单，这可以说明它们的受欢迎程度。我们将在本节末尾的创新练习中查看此类系统的一些示例。

DNA 计算机。DNA 是驱动生物发展进程的核心动力，现代分子生物学已经提供了对 DNA 的离散变化的理解。1994 年，伦纳德·阿德尔曼（Leonard Adelman）想象利用这些变化来实现计算并模拟图灵机。实验证实了这种方法的有效性：可以用 DNA 构建计算机！当然，一些自然生物过程是否以相同的方式运作完全是另一个问题。

795

| 模型                                | 描述                                        |
|-----------------------------------|-------------------------------------------|
| 20 世纪初                            |                                           |
| 半图埃系统<br>( Thue, 1910 )           | 字符串替换规则，可以按任意顺序使用                         |
| 正式波斯特系统<br>( Post, 1920s )        | 字符串替换规则，旨在证明来自一组公理的数学陈述                   |
| 20 世纪中                            |                                           |
| 拉姆达演算<br>( Church, 1936 )         | 一种定义和操作函数的方法<br>( 函数编程语言的基础，如 Lisp 和 ML ) |
| 图灵机<br>( Turing, 1936 )           | 在无限纸带上读写的有限自动机                            |
| 波斯特机<br>( Post, 1936 )            | 带队列的图灵机                                   |
| 递归函数<br>( Gödel 等人, 1930s )       | 定义对自然数计算的函数                               |
| 无限制文法<br>( Chomsky, 1950 )        | 用于描述自然语言的字符串替换规则                          |
| 2D 元胞自动机<br>( von Neumann, 1952 ) | 二值类型的二维数组，每个元素依据规定的<br>规则随邻居值的变化而变化       |
| 马尔可夫系统<br>( Markov, 1960 )        | 字符串替换规则，按预先指定的顺序使用                        |
| 通用计算模型                            |                                           |

|                                    |                                                                        |
|------------------------------------|------------------------------------------------------------------------|
| 霍恩子句逻辑 (Horn, 1961)                | 基于逻辑的定理证明系统 (Prolog 编程语言的基础)                                           |
| 双寄存器 DFA (Minsky, 1961)            | DFA 加两个计数器 (每个计数器存储一个整数, 机器可以递增、递减, 如果为零则测试)                           |
| 双栈图灵机 (Shepherson/Sturgis, 1963)   | 图灵机加两个下推栈                                                              |
| 游戏人生 (Conway, 1960s)               | 特定的 2D 元胞自动机                                                           |
| 指针机                                | 有限多个寄存器加上按链表访问的内存                                                      |
| 随机存取机                              | 有限多个寄存器加上通过索引访问的内存                                                     |
| <b>20 世纪末</b>                      |                                                                        |
| 编程语言                               | Java, C, C++, Python, Matlab, R, Fortran, Lisp, Haskell, Basic, Prolog |
| Lindenmayer 系统 (Lindenmayer, 1976) | 字符串替换规则, 并行应用 (用于模拟植物生长)                                               |
| 台球计算机 (Fredkin 和 Toffoli, 1982)    | 难以区分的台球在平面内移动, 相互之间产生弹性碰撞和内部障碍                                         |
| 粒子计算机                              | 粒子通过空间传递信息 (当粒子碰撞时发生计算)                                                |
| 1D 元胞自动机 (Cook, 1983)              | 二值类型的向量, 根据特定规则改变, 取决于其邻居的值                                            |
| 量子计算机 (Deutsch, 1985)              | 通过量子态的叠加计算<br>(基于 Feynman 在 20 世纪 50 年代的工作)                            |
| 广义移位图 (Moore, 1990)                | 单个经典粒子在由抛物面镜制成的三维势阱中移动                                                 |
| DNA 计算机 (Adelman, 1994)            | 通过 DNA 链的生物操作进行计算                                                      |

#### 通用计算模型 (续)

对普遍性概念的了解导致了观点的戏剧性转变。似乎我们在自然界中观察到的任何类似于计算机的东西实际上都是计算机。尽管已经进行了数十年的尝试, 但白图灵以来开发出来的每一个至少与图灵机一样强大 (可以模拟图灵机)、合理的计算模型, 也被证明没有图灵机强大 (因为它可以用图灵机模拟)。

这种知识很重要, 因为我们可以使用像这样简单的图灵机作为证明计算事实的通用模型。可以用完全数学严谨性证明图灵机, 使得我们也可以证明我们经常使用的计算机。这是计算机理论中最重要的一点, 下一节, 我们从这一点开始讲解。关于图灵机的知识是否也适用于自然世界本身的问题是一个哲学问题, 值得深思。

[797]

### 问答环节

**问:** 真的有可能建立一个模拟传统计算机的图灵机, 让它就像我的笔记本计算机中的微处理器一样工作吗?

**答:** 是的。这正是邱奇-图灵论题所说的。明斯基的经典著作就描绘了这种图灵机的蓝图。

**问:** 图灵机模型是不是比真正的计算机更强大, 因为图灵机有无限的纸带, 而真正的计算机只有有限内存?

**答:** 从技术意义上讲, 是的。可以使用一个庞大的 DFA 模拟真实计算机。但是, 这个 DFA 需要有至少  $2^{1\,000\,000\,000}$  个状态才能为具有 1GB 内存的计算机建模! 虽然真正的计算机只能访问有限的内存, 但实际上, 如果把互联网也算在内的话, 则该数量几乎是无限的。同样, 如果你认为宇宙中可访问的数据位数是有上限的, 那么你必须承认图灵机是纯粹虚构的模型, 是无法实现的。

**问:** 图灵机能模拟每种类型的计算吗?

**答:** 图灵机专为判定语言和计算函数而设计, 对于其他一些类型的计算不一定能很好

地完成。例如，生成随机数、控制自动驾驶汽车或制作蛋奶酥。要对这些类型的计算进行建模，你需要将图灵机连接到合适的外围设备。

问：为什么不设计一台比图灵机更强大的机器呢？

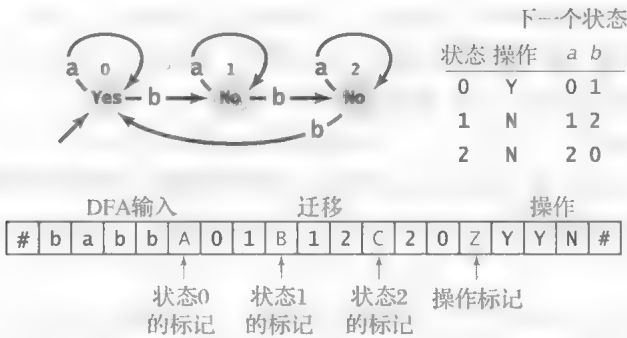
答：人们已经这样做了。理论上说，超级通用计算设备是一种功能更加强大的抽象模型，能够计算图灵机所不能计算的任务。实现这种模型的一种方法是存储连续值而不是离散的符号。然而，我们无法确定：自然界中是否存在连续值？如果存在的话，自然界中是如何处理这些连续值的？

798

创新练习

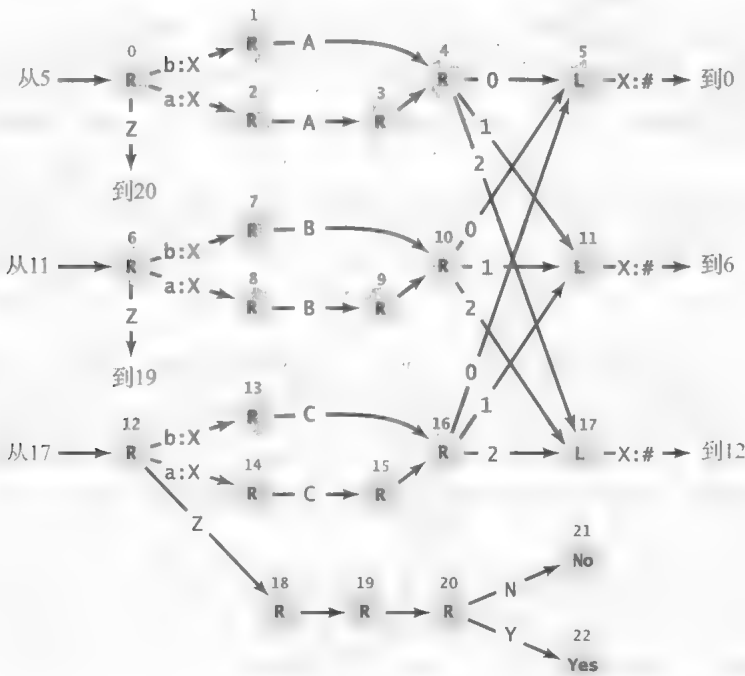
5.3.1 通用虚拟 DFA（表示形式）。开发一个适用于图灵机纸带的 DFA 表示。

答案：从 DFA 输入开始，然后是一个四行的状态表（表头占据了一行），分别标记为符号 A（对于状态 0）、B（对于状态 1）和 C（对于状态 2），每行后跟两个数字（0,1 或 2）给出两个可能的 DFA 输入符号的下一个状态。在状态表之后是标记符号 Z，后跟三个符号，用于为每个状态指定操作，它们的值是 Y 或 N。



5.3.2 通用虚拟 DFA。开发一个图灵机，可以模拟任何给定的三态 DFA 在任意给定输入上的操作。

答案：

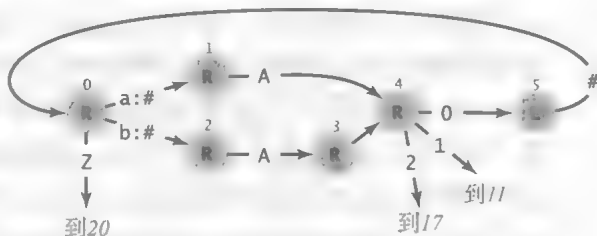


799



要了解此图灵机的运行方式,请考虑前六个状态,如右图所示。这些状态对应于 DFA 状态 0 (每个 DFA 状态有六个类似的状态)。

- 状态 0 向右扫描寻找 DFA 输入符号,如果没有这样的符号,那么就会扫描到标记 Z。
- 状态 1 和 2 扫描寻找标记 A; 它们之后是 DFA 状态 0 的迁移。如果输入符号是 b, 则状态 3 将跳过下一个符号。
- 状态 4 将会发生状态迁移,迁移到与状态 5, 或者是与 DFA 状态 1 和 2 对应的机器中的状态 (11 和 17)。
- 状态 5 向左扫描到输入的最左端,准备读取下一个符号。



当遇到 Z 标记时,表示 DFA 输入已读完,状态 20 读取状态 0 的动作并选择进入 Y 或 N 状态。

对应于 DFA 状态 1 和 2 的图灵机部分与对应于 DFA 状态的六个状态相同,只是它们还可以分别扫描 B 和 C 标记,以在状态表中找到它们的行,并且它们在状态 19 和 18 中分别扫描到 Z 标记之后会跳到相应的正确状态。请注意,此图灵机没有 H 状态,并且不会以无限循环结束,因为每个 DFA 都会要么接受要么拒绝其输入字符串。因此,对于任何 DFA 和任何输入字符串,这个图灵机可以判定给定的 DFA 是否接受给定的输入。

- 5.3.3 通用虚拟 DFA (跟踪)。对于练习 5.3.2 中的图灵机,假设 DFA 用于描述 b 的数量是 3 的倍数的语言 (见练习 5.3.1), 输入为 babb, 请给出图灵机的追踪。并解释如果输入扩展成 babbb, 跟踪的信息会是什么样的。

部分答案: 如下。

| # | b | a | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 从状态0开始          |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-----------------|
| # | # | a | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 0-2 向右扫描到b      |
| # | # | a | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 2-3 向右扫描到A      |
| # | # | a | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 3-4 跳过          |
| # | # | a | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 4-11 到达DFA的状态1  |
| # | # | a | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 11-6 向左扫描直到#    |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 6-7 向右扫描直到a     |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 7-10 向右扫描直到B    |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 10-11 到达DFA的状态1 |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 11-6 向左扫描直到#    |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 6-8 向右扫描直到b     |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 8-9 向右扫描直到B     |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 9-10 跳过         |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 10-17 到达DFA的状态2 |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 17-12 向左扫描直到#   |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 12-14 向右扫描直到b   |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 14-15 向右扫描直到C   |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 15-16 跳过        |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 16-5 到达DFA状态0   |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 5-0 向左扫描直到#     |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 0-20 向右扫描直到Z    |
| # | # | b | b | b | A | 0 | 1 | B | 1 | 2 | C | 2 | 0 | Z | Y | Y | N | # | 0-20 在状态22时接受   |

5.3.4 通用虚拟 DFA (模拟)。创建一个文本文件,以图表形式描述练习 5.3.2 中的通用 DFA 图灵机。从本书网站下载 TuringMachine.java 和 Tape.java,按照练习 5.2.7 和练习 5.2.8 中的描述进行修改,以指定的输入运行图灵机得到一个跟踪信息,检查练习 5.3.3 的答案对不对。

5.3.5 受限制的图灵机。纸带头只能在一个方向上移动的图灵机不是通用的,指出一种不被该机器识别的语言来证明如上所述。

5.3.6 半图埃系统。请考虑以下字符串替换规则集:

```
a -> c
aa -> b
ab -> abc
```

假设可以任何顺序应用这些替换规则,一次只能应用一个,以达到将一个字符串转换为另一个字符串的目的。那么,是否有可能将 aababca 变成 bbccbcc?

答案:可以, aababca -> aabcabca -> bbcabca -> bbccbca -> bbccbcc。

5.3.7 图埃系统。请考虑以下字符串替换规则集:

```
ac <-> ca
ad <-> da
bc <-> cb
bd <-> db
eca <-> ce
edb <-> de
cdca <-> cdcae
aaa <-> aaa
daa <-> aaa
```

与半图埃系统类似,可以任何顺序使用上述规则,一次应用一个,只是这次的规则是对称的,可以在任一方向上应用,目标仍然是将一个字符串转换为另一个字符串。那么,是否有可能将 abcacccddaa 转换为 aaa?

5.3.8 马尔可夫系统。在马尔可夫系统中,以字符串开始,运用字符串替代规则,直到再无规则可用。当最后结果为 1 时,原始字符串可以接受;否则拒绝。例如如下马尔科夫系统:

```
ab -> 1
a1b -> 1
```

字符串 aaabbb 被此系统接受,因为我们可以应用第一个规则来获取 aa1bb,然后应用第二个规则两次得到 1。相反, aabbabb 被拒绝,因为在应用第一个规则两次之后我们将获得 a1b1b,然后是第二个规则,给了 11b,我们被卡住了。设计马尔可夫系统,识别字母表 {a, b} 组成的所有回文序列。

5.3.9 波斯特系统。给定由变量(大写字母)和符号(其他符号)组成的公理和替换规则列表,不确定地应用替换规则以获取字符串。例如,从公理  $1 + 1 = 11$  和替换规则开始。

```
X+Y=Z -> X1+Y=Z1
X+Y=Z -> X+Y1=Z1
```

你可以连续三次应用第一条规则得到  $1111 + 1 = 11111$ ,然后连续两次应用第二条规则得到  $1111 + 111 = 1111111$ ,然后再次应用第一条规则得到  $11111 + 111 = 11111111$ 。描述此波斯特系统生成的语言。

5.3.10 匹配的括号。设计一个波斯特系统,生成所有完整匹配的括号字符串,如 ()、()()、((()、(((()()))))等。

5.3.11 Lindenmayer 系统。Lindenmayer 系统(L 系统)的工作原理是从一个初始字符串开始,然后并行应用替换规则。比如规则是用 FLFRRFLF 替换所有出现的 F,如果初始字符串是 FRRFRRF,则在一次迭代之后我们得到 FLFRRFLFRRFLFRRFLFRRFLFRRFLF。

使用这些规则开发一个 Turtle (程序 3.2.4) 的客户程序, 用来绘制科赫曲线 (参见 3.2 节)。将 F 解释为向前画一条线, L 向逆时针旋转  $60^\circ$ , R 向顺时针旋转  $60^\circ$ 。然后, 第  $n$  次迭代之后的字符串是一个  $n$  阶的科赫曲线。编写一个 Java 程序 Lindenmayer, 它接受命令行参数  $n$  并打印生成  $n$  阶的科赫曲线的指令。提示: 使用方法 `String.replaceAll()`。

5.3.12 方形科赫曲线。使用上述<sup>⊖</sup> Lindenmayer 系统产生方形科赫曲线: 从字符串 F 开始, 然后用 FLFRFRFFLFLFRF 不断地替换所有出现的 F。

803

5.3.13 Bracketed L 系统。假设一个 Lindenmayer 系统, 以字符 F 开头并重复用 FFR [RFLFLF] L [LFRFRF] 并行替换 F。将 L 和 R 解释为  $22^\circ$  旋转, 将符号 “[” 和 “]” 解释为向栈中压入或弹出海龟的当前状态 (位置和角度)。括号避免了图片由单线组成的情况。

5.3.14 希尔伯特曲线。使用以下 L 系统创建希尔伯特曲线: 从字符 A 开始, 然后不断地用字符串 LBFRAFARFBL 替换 A、用字符串 RAFLBFBLFAR 替换 B。同时应用这两个替换规则。第一次迭代结果为 LBFRAFARFBL, 第二次结果为:

LRAFLBFBLFARFRLBFRARFBLFLBFRARFBLRFRALBFLFARL

当使用海龟绘图显示曲线时, 将 L 解释为“向左转  $90^\circ$ ”, 将 R 解释为“向右转  $90^\circ$ ”, 将 F 解释为“向前移动”, 忽略 A 和 B。

5.3.15 龙形曲线。使用以下 L 系统创建龙形曲线 (参见练习 1.2.35): 以字符串 FA 开始。然后反复用 ALBFL 替换 A、用 RFARB 替换 B。如练习 5.3.14 所述, 将字母解释为海龟绘图的指令。忽略 A 和 B, 前 3 次迭代的结果分别是 FLFL、FLFLLRFRFL 和 FLFLLRFRFLLRFLFLRRFRFL。

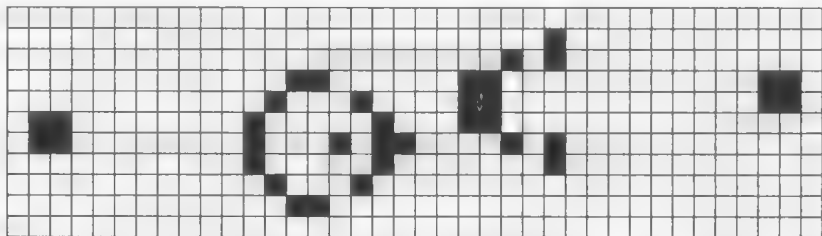
5.3.16 标签系统。编写一个程序, 读取二进制字符串并应用以下 (00,1101) 标记系统: 如果第一位为 0, 则删除前 3 位并在尾部追加 00; 如果第一位为 1, 则删除前 3 位并在尾部追加 1101。只要该字符串不少于 3 位, 就不断重复上述过程。尝试确定以下输入是停止还是进入无限循环: 10010,1100100100100100100。考虑使用队列。

5.3.17 波斯特机。假设标签系统是通用的, 通过描述如何使用波斯特机模拟任何标签系统, 以此来证明波斯特机 (带队列的 DFA) 也是通用的。

5.3.18 双栈 DFA。假设波斯特机是通用的, 通过描述如何模拟具有两个栈的队列来显示具有两个栈的 DFA 也是通用的。

804

5.3.19 合成滑翔机。初始化康威的生命的游戏 (参见练习 2.4.20) 为一个  $50 \times 50$  网格, 下图显示的是网格的左上角。这种模式被称为滑翔机枪 (glider gun), 它可以产生新的滑翔机。它是由 R. Gosper 在 1970 年发明的, 这是一种可以无限增长的模式, 这个发明赢得了康威的挑战赛 (康威曾推测, 没有任何图案能够无止境地生长。他曾拿出 50 美元作为奖品, 奖励第一个能够在那年证明这个命题或者将其证伪的人——译者注)。你可以认为滑翔机生成器正在传输某种信息。它被 Paul Rendell 在 2011 年用作实现通用图灵机模拟器的基本构建块之一。



805

⊖ 原文错误。——译者注

## 5.4 可计算性

现在我们已经对算法是什么有了一个清楚的认识——算法就是图灵机。我们可以通过对图灵机的证明来探究计算的本质。邱奇-图灵论题告诉我们，当我们有任意一种可用的计算设备，且在这个设备上能够有一个有限的、确定的且有效的方法能够解决某个问题，我们就期望能够构建一个图灵机来解决这个问题。该论点反其道而行之更为成立：如果我们能够证明判定一种语言或者计算函数的图灵机不存在，那么我们便认为这个任务在物理上是不可执行的。图灵论文的中心焦点是如何在图灵机可以解决的问题和图灵机不能解决的问题中划分出一条清晰的界限。在本节中，我们将介绍图灵机模型中一个非常重要的结论：这个结论可以用来证明不可判定的语言和不可计算的函数是存在的——在这种情况下，不存在能完成这些工作的图灵机。我们将这些问题称为不可解问题，因为我们认为无论是现在还是将来都不存在可以完成这些工作的图灵机，即没有解决这些问题的算法。

不可解性是问题的一个重要特性。它表明的不是对于这个问题的算法科学家们尚未找到解决方案，而是不可能找到。了解问题的不可解性是很重要的，因为这能让你知道你试图解决的是一个无法解决的问题，这样你就可以避免浪费时间和精力来解决它，转而解决更容易解决的问题。在过去的一个世纪里，很多人都为解决某些问题努力工作，但这些问题在后来都被证明是不可解的。那些不了解图灵理论的人就像在黑暗中工作，他们有可能会浪费大量的精力来完成不可能完成的任务。

**背景：希尔伯特计划** 20 世纪初期，当时杰出的数学家大卫·希尔伯特提出了一个雄心勃勃的计划，这个计划的目的是为了解决逻辑和数学中的一些最基本的问题。他和他的同事挑战了一个难题，对下列三个陈述进行严格证明：

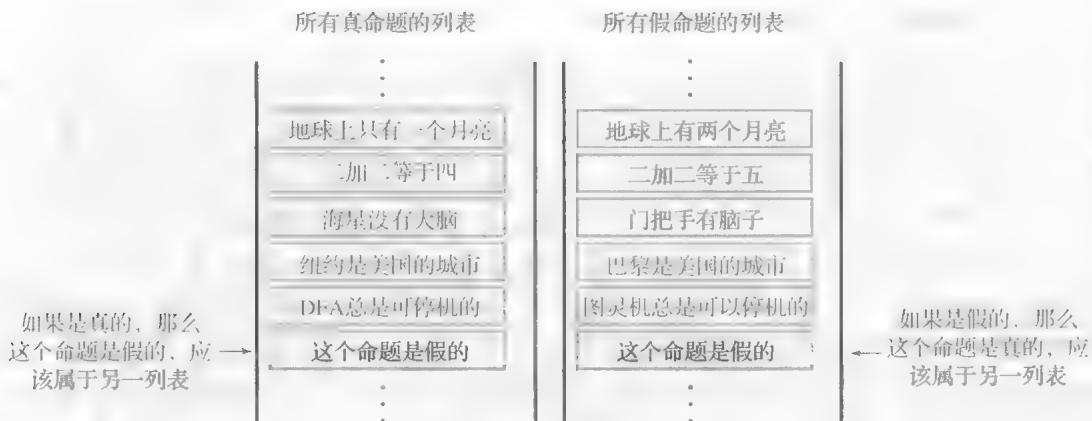
- 数学是一致的：不可能同时证明一个陈述和它的反论，就像你不能证明  $1=2$ 。
- 数学是完备的：如果一个数学陈述是真的，那么就可以证明它是真的。
- 数学是可确定的：对于任何数学定理，都可以用公理逐步推导进行证明。

806

这个计划在 20 世纪中最重大的进展发生在 1930 年，当时库尔特·哥德尔 (Kurt Gödel) 以最令人惊讶的方式解答了前两个陈述：他证明了任何公理系统（能够对算术建模）都不可能是既一致又完备的。这个发现动摇了整个学术界的基础，将人们长期以来相信的东西置于混乱之中，并使得人们开始对基础数学进行广泛的研究。

究竟怎样才能算是一个“逐步推导的过程”呢？究竟什么是“数学定理”呢？“数学”本身又是什么呢？图灵机成功地解决了这些问题中的第一个，并为希尔伯特计划的最终解决方案奠定了基础：数学不可能是既一致又完备的，并且也是不确定的，因为存在无法证明的定理。在本节中，我们将在计算问题的范畴中研究这些概念。

**示例：说谎者悖论** 作为示例，我们来研究一下说谎者悖论 (liar's paradox)，这个悖论可以追溯到古希腊的哲学家。假设我们的目标是将所有命题分类为真或假，例如，我们将陈述“二加二等于四”和“地球有一个月亮”分类为真，将陈述“二加二等于五”和“地球有两个月亮”分类为假。这似乎是个合理的目标，但是当我们对以下命题进行分类时会遇到不可逾越的障碍：“这个陈述是假的。”如果我们将它分类为真，那么“这个陈述是假的”就是假，所以我们需要将它分类为假。如果我们将它分类为假，那么“这个陈述是假的”就是真，所以我们需要将它分类为真。任何一种情况都会导致矛盾。



说谎者悖论

摆脱这种自相矛盾的唯一方式是承认原本的前提一定是假的。这种证明技巧被称为反证法：如果一个假设最终推出来的是一个荒谬的结论，那么这个假设一定是假的。在说谎者悖论的例子中，我们的假设是可以将所有命题都归类为真或假。而“这个命题是假的”这个命题，无论将它归为哪一类，最后都会产生矛盾，所以我们最开始的假设一定是假的。换句话说，不可能把所有的命题都分类为真或假。起初这个结论被当成一个微不足道的论点，但是实际上这个结论有着重大意义，这样的论点可以作为证明其他有趣事实的基础。

注意这不仅仅是一个数学不一致的例子，也不仅仅是一个悖论，这个证明确立了数学学科中的一个事实：不可证明的问题是存在的。

**停机问题** 依据图灵论文，下面我们证明停机问题是无法解决的，通过这个证明我们可以证明存在一个无法解决的问题。停机问题非正式地描述为：给定一个程序和它的输入，确定这个程序在给定输入上运行时是否会停机。由于现在的计算机在实际中并不会经常进入停机状态，所以“停机”这个术语在这里指代程序不会进入无限循环。例如，我们考虑一个 Java 函数，当这个 Java 函数将控制权返回给调用者，而不是进入无限循环时，我们认为这个 Java 函数“停机”了。

由于所有的程序员都体验过程序进入无限循环带来的不良影响，所以在程序运行前检查是否会发生无限循环是非常有必要的。这也是判断初学者编写的程序质量如何的重要依据之一。再举一个例子，我们思考一下软件公司的质量控制部门面临的挑战：这些公司都希望能证明它们的软件不会导致你的移动设备无故死机。我们希望有一个程序可以检查在给定输入上运行的程序是否会进入无限循环，但是图灵的证明告诉我们：开发这样一个程序是不可能的。

假设 UTM 是一个将图灵机和图灵机的输入作为输入的程序，这个程序可以模拟该图灵机的操作。停机问题实际上是询问是否有这么一个 TM，这个 TM 执行了一个看起来很简单任务，即确定给定的 TM 是否会在指定的输入上进入停机状态。

**Java 的公式化证明。**对于所有的问题形式，我们都可以用 Java 来重新表示停机问题。虽然依据图灵机的定义来重新表述相关证明过程也不难，但是对于具有编程经验的人来说，Java 表达式更为直观。所以我们将停机问题表示如下：是否存在一个表达式  $\text{halts}(f, x)$ ，这个表达式以函数  $f$  和输入  $x$  作为参数（二者都编码为字符串），能够确定调用  $f(x)$  是否陷入无限循环。特别的， $\text{halts}(f, x)$  必须具有以下形式：

```
public static boolean halts(String f, String x)
{
 if (/* 一些极其聪明的判断 */) return true;
 else return false;
}
```

为了解决停机问题，halts() 本身不能进入无限循环，而且它必须为每个函数  $f$  (这个函数只有一个 String 参数) 和每个输入  $x$  提供正确的答案。

与 UTM 一样，你可以将 Java 视为一个程序，它将你的程序及其输入 (编码为字符串) 作为它的两个参数，然后运行你的程序来产生期望的计算结果。是否有这么一个简单的程序，这个程序同样需要相同的两个参数，就能够确定是否会陷入无限循环？

一个典型例子。为了弄明白为什么这是一个很艰巨的任务，请你考虑以下两个函数，它们之间只有一个字符不同。

```
public static void f(int x) public static void g(int x)
{
 while (x != 1) {
 if (x % 2 == 0) x = x / 2; while (x != 1)
 else x = 2*x + 1; {
 if (x % 2 == 0) x = x / 2;
 else x = 3*x + 1;
 }
 }
```

当且仅当  $x$  不是 2 的正整数次幂时，左边的函数进入无限循环；而右边的函数实现了我们在练习 2.3.29 中遇到的克拉茨序列 (Collatz sequence)，这个函数的情况不太清晰，因为没有人知道这个函数是否可以对于任意的  $x$  能够停机。对于任何给定的  $x$ ，我们需要等待多久才能判定它处于一个无限循环中呢？我们可以运行一下程序来看看会发生什么。如果它一直不停地运行的话，我们该怎么办呢？也许我们再继续运行一会儿它就会停机。一般来说，我们无法确切地知道一个程序会运行多久。就算是数学家已经证明了在输入值小于  $10^{300}$  的情况下会终止，但是我们也总会找到一个更大的值，而这个值是否能够停机还是需要重新测试 (见练习 5.4.7)。这是一个极端的例子，但是它强调了一个事实，这个事实是判断一个给定的程序是否会停机并不是一件简单的事。一步一步的模拟程序的操作比确定一个程序是否会进入无限循环更容易。事实上，证明程序是否会进入无限循环是一个不可能完成的任务。你可能会觉得这个结论难以接受，接下来我们会证明它。

809

证明不可解性。有的问题是不可解的，这可能会粉碎你对于计算的一些根深蒂固的想法。所以我们建议你多看几次证明过程，直到你对这个观念不再感到惊讶并坚信它是正确的。这是 20 世纪最重要的思想之一。

**定理 (图灵, 1937):** 停机问题是不可解的。

**证明过程:** 为了证明这个定理是矛盾的，我们假设存在一个前面描述的停机表达式  $\text{halts}(f, x)$ 。我们的第一步是创建一个新的表达式  $\text{strange}()$ ，这个表达式将一个函数  $f$  (编码为一个字符串) 作为输入，然后调用  $\text{halts}()$ ，如下所示：

```
public static boolean strange(String f)
{
 if (halts(f, f))
 while (true) /* 无限循环 */ ;
}
```

调用  $\text{halts}(f, f)$  来检查程序是否停机可能很奇怪，因为函数将它自己作为输入了。我们这么做只是为了证明技巧需要。但是这实际上并不奇怪。例如，想象一下编译器的设计者希望检查编译器编译自己的时候是否会进入无限循环。

那么 `strange()` 实际上干了什么呢？回想一下，如果  $f(x)$  停机则 `halts(f, x)` 返回 `true`，如果  $f(x)$  不停机则返回 `false`，这里的术语“停机”表示“不会进入一个无限循环”。因此，测试下列代码：

- 如果  $f(f)$  停机，那么 `strange(f)` 就不会停机。
- 如果  $f(f)$  不停机，那么 `strange(f)` 停机。

现在我们来执行关键的一步：当我们把函数自身（编码为一个字符串）作为输入来调用 `strange()` 时发生了什么？也就是说，我们将上面两个陈述中的  $f$  替换为 `strange` 时，就会得到两句（奇怪的）陈述：

- 如果 `strange(strange)` 停机，那么 `strange(strange)` 就不会停机。
- 如果 `strange(strange)` 不停机，那么 `strange(strange)` 停机。

这两个陈述都是既矛盾又荒谬的，这让我们得出这样一个结论：我们假设存在的停机函数 `halts(f, x)` 不存在。也就是说，停机问题是无法解决的！

如果你觉得这只是一个符合逻辑的技巧，请你再次阅读这个证明，然后尝试完成练习

**810** 5.4.7. 停机问题的不可解性是计算的一个有重大意义的陈述，具有重要的实际意义。

**归约法** 停机问题不仅非常有趣，而且它在很大的范围内都十分重要，因为我们可以用它来证明其他重要问题是无法解决的。用于此目的的技术被称为归约法（problem reduction）：

**定义：**当满足如下条件时，我们可以把一个问题 A 归约成另一个问题 B：

如果给出 B 的一个程序，可以采用下列方法对 A 中的任意实例  $a$  求解：

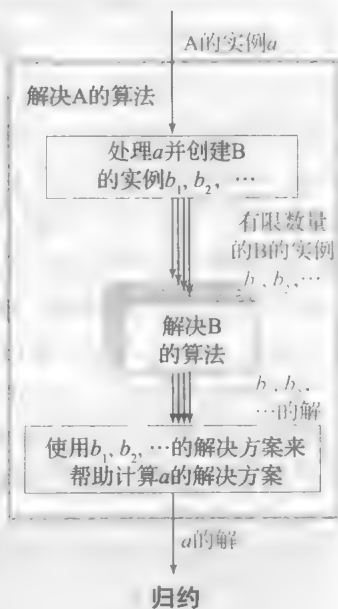
- 处理  $a$  并创建 B 的实例  $b_1, b_2, \dots$ 。
- 使用 B 的程序来获得  $b_1, b_2, \dots$  的解。
- 使用  $b_1, b_2, \dots$  的解决方案来帮助解决  $a$ 。

这是一个简单的概念，但是在最开始的地方有点混乱，因为最开始 B 的实例可能有多个。实际上，在我们的所有例子里面都只使用了 B 的一个实例。而且，我们使用的归约是与不可解性相对的。我们首先把问题 A 看作停机问题，然后证明问题 B 的一个解决方案可以用于解决停机问题。这就意味着 B 也是不可解的，因为停机问题是不可解决的。

**整体性。**我们将要研究的第一个例子是整体性问题。我们是否可以编写这么一个程序，它将一个函数作为输入，并且确定对于任意输入它是否会进入无限循环。例如，解决了这个问题就能解决前文中用于表达克拉茨猜想的 `g()`。任何软件公司都希望有这么一个程序来证明它的产品不会进入无限循环。但是我们可以用归约法证明这个问题是不可解的。

**命题 A：**整体性问题是不可解决的。

**证明过程：**将整体性问题看作“问题 B”，假设我们有一个 Java 函数 `alwaysHalts()` 来证明这个问题：`alwaysHalts()` 将任意函数  $f$  作为一个输入参数，如果  $f(x)$  对于所





有  $x$  停机则打印 Yes, 如果  $f(x)$  对于某个  $x$  进入无限循环则输出 No。现在有一个使用 `alwaysHalts()` 来解决停机问题的方法: 给定任何函数  $f$  和参数  $x$ , 定义一个不带参数的函数  $g()$ , 这个函数仅仅是调用  $f(x)$ 。如果  $f(x)$  停机则调用 `alwaysHalts(g)` 打印 Yes, 否则打印 No。也就是说它解决了停机问题。这是一个矛盾的说法, 所以我们存在一个 `alwaysHalts()` 的假设肯定是错误的。也就是说整体性问题是无法解决的。

811

总之, 我们通过把停机问题归约成整体性问题, 从而证明整体性问题是不可解的。如果我们能解决整体性问题, 那么我们就可以解决停机问题。由于停机问题是不可解的, 所以整体性问题也是不可解的。

将问题 A 看作任意不可解的问题, 这样的论述同样奏效。通过仔细研究这个例子和下小节中的例子, 你能更好地感受这个证明过程。如果你没有深入研究数学, 那你最好稍微浏览一下证明的过程, 并且在第一次阅读的时候就尝试理解这个结论。之后你可能会更加仔细地研究这个证明, 因为它们与典型的数学证明相比其实很简单。

等价程序。我们是否可以编写这么一个程序, 这个程序将两个函数作为输入并确定这两个函数是否等价 (即这两个函数对于给定的输入会产生相同的输出)? 同样的, 任何软件公司也会想要这样的一个程序。我们也同样可以通过归约法来证明这个问题是不可解的。

**命题 B:** 等价程序问题是不可解的。

证明过程: 假设我们有一个 Java 函数 `areEquivalent()`, 这个函数将任意函数  $f$  和  $g$  作为参数, 如果它们是等价的则输出 Yes, 否则输出 No。给定任意的一个 Java 函数  $f()$ , 我们调用 `areEquivalent(f, h)`, 其中  $h$  是一个刚刚返回的函数。这样问题又变成了确定  $f()$  是否对于任意输入都不会进入一个无限循环, 这是一个整体性问题。

总之, 我们通过把整体性问题归约成等价问题, 从而证明等价问题是不可解的。如果我们能解决等价问题, 就可以解决整体性问题。因为整体性问题是不可解的, 所以等价问题也是不可解的。

莱斯定理。这些属性只是冰山一角。当一个程序的任何输入 / 输出的行为属性是非平凡的, 也就是说, 这些属性是某些程序具备而非所有程序具备的, 我们就把这样的属性称为一个程序的功能属性。亨利·莱斯 (Henry Rice) 在 1951 年的博士论文中证明了下面的定理:

**定理 (莱斯, 1951):** 确定一个给定的程序是否具有某个特定功能属性, 该命题不可解。

812

这个定理具有极其广泛的适用性。一个 Java 程序可以在标准输出上写超过  $10^{1000}$  个符号吗? 它能输出什么吗? 它会基于不只一个参数值而进入无限循环吗? 如果它所有的参数都是合法的, 它会停机吗? 一个没有参数的 Java 程序会停机吗? 你可以很轻松地添加几十个属性到这个列表中。许多听起来很自然的问题都与程序的功能特性有关。

不可解性和莱斯定理从一个非常实际的角度出发, 为我们理解“为什么确保软件系统的可靠性如此困难”提供了基础。这就像我们想编写一个程序来确保软件具有很多我们想要的属性一样, 这是不可能做到的。了解这个事实的人会比不了解这个事实的人在计算领域更成功。

**更多不可解问题的例子** 不可解性不仅限于那些处理程序的程序（这些程序很重要）。自从图灵提出这个概念以来，经过了数十年，研究人员一直在用归约法来大量地扩展不可解问题的数量，并在数学、科学和工程的各个领域都有应用。某个已知的不可解问题被归约成一个新问题，若新问题可以解决最终意味着停机问题可以解决，所以新的问题被证明是不可解的。我们在后面的表格中引用了许多重要又有趣的例子，接下来我们会更详细地讨论其中的一些例子。

**邮政通信问题。**下面的问题涉及的内容是一些写在卡券上的字符串，这些字符串最早是由埃米尔·波斯特（Emil Post）在1940年分析的。一个邮政通信系统就是一个已定义的卡券类型的集合。每种类型由两个字符串修饰，一个写在卡券的顶部，另一个写在卡券的底部。例如，右边的示例展示了四种卡券类型，第一种顶部是BAB，底部是A；第二种顶部是A，底部是ABA，等等。我们要解决的问题是，是否可以将卡券（使用任何数量的每种类型的卡券）排成一列，使得顶部和底部的字符串相同。在我们的例子中，答案是肯定的，如图下部的解决方案所示：一张类型为1的卡券跟着一张类型为3的卡券，然后是一张类型为0的卡券，然后是一张类型为2的卡券，最后是第二个类型为1的卡券，这样顶部和底部的字符串都是ABABABABA。

[813]

卡片类型

|          |          |         |         |
|----------|----------|---------|---------|
| BAB<br>A | A<br>ABA | AB<br>B | BA<br>B |
| 0        | 1        | 2       | 3       |

解决方案

|          |         |          |         |          |
|----------|---------|----------|---------|----------|
| A<br>ABA | BA<br>B | BAB<br>A | AB<br>B | A<br>ABA |
| 1        | 3       | 0        | 2       | 1        |

一个邮政通信系统

另一个例子表明这并不总是可能完成的，如左图所示。为什么在这种情况下没有解决方案呢？在开始的时候，解决方案中最左边的卡的顶部和底部的最左侧符号必须是一致的，但是没有任何一张卡的顶部和底部的最左侧符号是相同的，所有没有方法能适当地排列这些

|          |          |         |         |
|----------|----------|---------|---------|
| A<br>BAB | ABA<br>B | BA<br>A | AB<br>B |
| 0        | 1        | 2       | 3       |

另一个邮政通信系统

卡。一般来说，像这么简单就对例子进行解释的方式可能不存在，找到解决方案可能是一个挑战。邮政通信问题主要是开发一个算法，这个算法可以确定对于任何给定的系统是否都存在一个解决方案。值得注意的是，这个问题是不可解的。这个问题被证明可以归约成许多涉及字符串的其他问题，而这些问题也是不可解的。

**最优数据压缩。**你可能已经尝试过使用数据压缩算法来减少图片或数据文件的大小，这样可以方便地共享或存储它。你的系统中使用的算法是经过几十年研究的产物，但是我们很自然会问：是否能将文件的大小压缩得更小。在形式上，这个想法可以被看作最优数据压缩（optimal data compression）问题：给定一个字符串，找到能够生成这个字符串的最短程序（以字符数量来衡量）。例如，曼德布洛特集合（Mandelbrot set）是一个由简单程序生成复杂图片的典型例子（见程序3.2.7）。如果你尝试压缩系统上的曼德布洛特集合中的高分辨率图像，你可能会取得一些进展，但是却无法压缩到代表整个程序的几百个字符那么小。是否有能够从图像中推断出程序的算法呢？这个问题其实是奥卡姆剃刀（Occam's Razor）问题的一种具体化表述——寻找与事实相符的最简单的描述方法。虽然有正式的方法来简洁地描述这一发现会很不错，但是这个问题是不可解的。

**优化编译器。**在学术界关于编程语言研究的领域中，最优数据压缩被称为“充分就业定理”，它表示没有一个编译器可以保证它有对所有程序都能够进行优化的能力。从莱斯定理出发我们可以得到一系列充分就业定理，很多我们希望编译器为我们解决的问题实际上是不可解的。例如，程序是否有未初始化的变量？程序是否有永远不会执行的“死代码”？改

变特定变量的值和特定点的值是否会影响计算结果？程序能产生一个给定的字符串作为输出吗？很多像这样的问题我们都希望编译器来帮助我们解决，但是它们做不到。

814

| 问题         | 描述                            |
|------------|-------------------------------|
| 处理程序的程序    |                               |
| 停机问题       | 一个给定的程序对于一个给定的输入是否会进入无限循环？    |
| 整体性        | 一个给定的程序对于任一输入是否会进入无限循环？       |
| 等价程序       | 两个程序是否会得出相同结果？                |
| 内存管理       | 一个给定的变量是否会被再次引用？              |
| 病毒识别       | 给定的程序是不是病毒？                   |
| 功能属性       | 一个程序是否有某个功能属性？                |
| 其他例子       |                               |
| 邮政通信问题     | 一个给定的字符串替换规则集合是否有效？           |
| 最优数据压缩     | 是否可以对一个给定的字符串进行压缩？            |
| 希尔伯特的第十个问题 | 给定的多元多项式是否有整数根？               |
| 定积分        | 给定的积分是否有封闭解？                  |
| 群论         | 一个有有限个表示的组是简单的、有限的、自由的还是可交换的？ |
| 动态系统       | 一个给定的动态系统是混乱的吗？               |

不可解问题的例子

815

希尔伯特的第十个问题。1900 年，大卫·希尔伯特在巴黎召开的国际数学家大会上提出了 23 个问题，作为将要到来的 21 世纪的挑战。希尔伯特的第十个问题是设计一个过程，根据这个过程，可以通过有限数量的操作来决定一个给定的多项式（有多个变量）是否有整数根。换句话说，是否可以将整数值分配给多项式的变量使得多项式的结果为零。例如，多项式  $f(x, y, z) = 6x^3yz^2 + 3zy^2 - x^3 - 10$  有整数根，因为  $f(5, 3, 0) = 0$ ，而多项式  $f(x, y) = x^2 + y^2 - 3$  没有任何整数根。这个问题可以追溯到两千年前的丢番图方程，它出现在物理学、计算生物学、运筹学和统计学等多个领域。当时并没有对算法进行严格的定义，因此没有考虑到不可解的问题。在 20 世纪 70 年代，希尔伯特的第十个问题以非常令人惊讶的方式得到了解决：在马丁·戴维斯 (Martin Davis)、希拉里·普特南 (Hilary Putnam) 和茱莉亚·罗宾逊 (Julia Robinson) 奠定的基础上，尤里·马季亚谢维奇 (Yuri Matiyasevich) 证明了这个问题是不可解的，所有应用了这个模型得出的实际解决方案都是不可解决的。例如，通过对这个问题进行归约，很容易得出旅行计划问题是不可解决的，这意味着针对航空公司发布的航班和票价的数据，不可能找到一个算法可以查询任意一个旅行计划（即指定起点和终点的一次飞行计划）的最佳路线信息（或者确定这个路径不存在）。

定积分。数学家和科学家现在都广泛地依赖计算机系统来帮助他们进行符号化的操作。这些系统用计算机来完成泰勒级数、多项式相乘、积分和微分等复杂计算任务。开发这样的计算机系统的一个关键挑战是定积分：是否对于每个定积分都能找到一个封闭解，这个封闭解仅由多项式和三角函数构成？许多人为了这个任务的算法努力工作了多年，但是现在已经从希尔伯特的第十个问题的归约中知道这个问题是不可判定的。

启示 对于计算原理解不够深入的人，往往会觉得我们可以用一个足够强大的计算机做任何事情。正如我们在本节的很多例子中看到的一样，这个假设毫无疑问是错的。不可解问题的存在对计算和哲学都有着深远的影响。这些问题让我们知道所有的计算机都受到计算方面的内在限制。我们必须认识到有些问题是不可解的，而无论这些问题多么重要。

816

除了它的实际重要性之外，不可解性（与邱奇-图灵理论一起）让我们有机会窥视自然界的计算规律，而且它还引发了一系列哲学问题。例如，如果邱奇-图灵理论适用于人脑，那么人类将无法解决像停机那样的问题。人类有可能会像计算机一样有根本的局限性。有没有哪个自然过程是普遍适用的？如果有的话，是否有因为不可解性而不能存在于自然界中的条件呢？是否有违反邱奇-图灵理论的自然过程呢？自从图灵工作的意义广为人知后，数学家和哲学家一直在挑战这些问题。人们普遍承认，他的论文是 20 世纪最重要的科学论文之一。



经 Nokia 公司许可转载

817

## 图灵理论的实践结果

### 问答环节

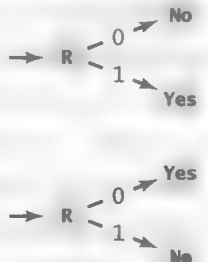
**问：**停机问题的不确定性告诉我们，我们不能写出一个 Java 程序来确定一个任意的程序是否在任意的输入上停机。但是我们可以写一个程序来确定一个特定的 Java 程序是否会在一个特定的输入上停机吗？

**答：**一个从业者可能会说我们可以写很多这样的程序，它们都确定会停机（如 HelloWorld.java）。一个理论家可能会说，你可以写两个程序，一个只是会打印 Yes，另一个只是会打印 No，这两个程序之中肯定有一个是正确的。当然，也有可能没有任何人知道正确答案是什么，但是这不能证明没有办法找到答案，因为这会产生悖论。对于选定的特定程序，如果它确实能够停机，那么我们可以运行它并获得一个程序会停机的证据。如果无法证明它是否会停机，那么它就一定是不能停机的，否则我们一定会找到能证明它会停机的证据。于是我们又可以用这个推论来作为它不会停机的证据！

**问：**克拉茨猜想是否是一个可判定的问题？

**答：**这里有同一个问题的另一个表述。西普塞（Sipser）用下面的方式表示它：如果在火星上有生命，那么令  $L$  为由所有由 1 组成的字符串组成的语言，否则  $L$  为由所有由 0 组成的字符串组成的语言。那么  $L$  是可判定的吗？排除中间的条件判断，即火星上要么有生命，要么没生命，根据以上论述，这个问题的答案为真。无论如何，右图的图灵机中肯定有一个是语言  $L$  的一个判别器，而且没有其他可能的判别器了。但事实上我们无法知道哪个判别器是我们应采用的。这个显而易见的悖论是基于语言的简单性。这个猜想是可判定的，但是我们不知道如何证明它是真的还是假的，我们也无法找到一个反例来证明。

**问：**是否能编写这么一个 Java 程序，这个程序可以解决不使用库函数且没有输入输出的 Java 函数的停机问题？

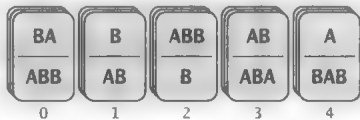


答：有的人可能会说这是有可能的，因为这样的程序只能使用有限的内存。但是这种说法认为我们的计算机都是 DFA，而且它们的性能也是天文数字。在这种情况下，我们描述的理论可能真的不适用，因为我们的理论的前提是每个程序都使用固定数量的资源，而这里编写的程序所需的所谓的固定数量已经要接近无穷大了。因此，你可以相信自己的直觉，我们讨论的模型抓住了机器的本质属性。

818

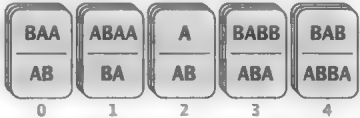
练习

- 5.4.1 假设在邮政通信问题中，每种类型的卡你最多只能使用一张。问题仍然是不可判定的吗？
- 5.4.2 找到以下邮政通信系统的两个解决方案，或者证明没有解决方案。



部分解：34012212。

- 5.4.3 找到以下邮政通信系统的两个解决方案，或者证明没有解决方案。



- 5.4.4 假设邮政通信系统的字母表中只有一个字母，那么我们只需要找到一个排列使得顶部和底部的字符串有相等数量的字符即可。在这种情况下设计一个算法来解决邮政通信问题。
- 5.4.5 假设存在下面这些替换规则：aba 替换为 bba，ba 替为 bbb，baa 替换为 aa，是否存在一些替换规则的使用序列（以任何顺序均可），可以将字符串 baababbba 替换为 ababbbabbba？（这是图厄词问题的一个例子，一般来说是不可判定的。）
- 5.4.6 修改练习 2.3.29 的解决方案中为了计算克拉茨问题给出的程序，使得这个程序可以使用 Java 的 BigInteger 类，以便它可以使用任意长度的整数进行计算。

819

创新练习

- 5.4.7 图灵机的停机问题。根据文中给出的图灵机的特征，重新证明停机问题的不可确定性。  
 以下每个练习都要求你证明给定的问题是不可解的。这些问题更倾向于那些研究数学的读者来解决，这些读者可能会享受通过归约来进行这些证明的乐趣。如果你不倾向于研究数学，这些问题对你来说仍然是值得阅读的，因为这可以加深你对不可解问题的了解。
- 5.4.8 自停机问题。我们是否可以编写这么一个程序，这个程序可以判定一个给定的、只有一个参数的函数将它自己作为输入时是否会停机。通过对一个停机问题输入相似的参数，我们可以证明这个问题是不可判定的。
- 5.4.9 穷忙问题。穷忙函数  $BB(n)$  有如下定义：对于一个基于二进制字母表有  $n$  个状态的图灵机，它有一个起始状态为空的纸带，穷忙函数即在保证这个图灵机仍会停机的前提下，图灵机可以在纸带上留下 1 的最大个数。证明  $BB(n)$  是不可计算的。提示：首先运行一个初始输入为空且有  $(m + n)$  个状态的图灵机，通过在这个图灵机上模拟有  $n$  个状态且输入大小为  $m$  的图灵机。然后，按照  $BB(m + n + 1)$  步骤运行有  $(m + n)$  个状态的图灵机。
- 5.4.10 空纸带停机问题。在练习 5.4.7 中，我们使用了一个把自己作为输入的图灵机来证明停机问题

的不可确定性。这种人为的自关联结构简化了证明。证明即使输入纸带最开始是空白的，停机问题也是不可确定的。提示：给出一个可以解决初始纸带为空的图灵机的停机问题的方法，展示它是如何计算穷忙函数  $BB(n)$  的。

5.4.11 非空。是否存在这么一个图灵机，它可以判定一个给定的图灵机接受的语言是否为空？证明这个问题是不可判定的。

5.4.12 常规性。是否存在这么一个图灵机，它可以判定一个给定的图灵机接受的语言是否是正则的？证明这个问题是不可判定的。

## 5.5 难解性

在前面的章节中，我们根据是否能被计算机解决来对问题进行了分类。在本节中，我们将重点放在我们可以解决的问题上，特别是如何确定解决这些问题所需要的计算资源。

在本书中我们研究了很多算法，这些算法普遍用于解决实际问题，而且它们消耗的计算资源的数量是合理的。大多数算法带来的实际效益是显而易见的，对于许多问题，我们都有很多高效的算法可以选择。不幸的是，许多在实践中出现的其他问题不能使用这些有效的解决方案。更糟糕的是，对于一大类这样的问题，我们甚至无法判断是否存在有效的解决方案。对于程序员和算法设计人员来说，这种情况是十分令人沮丧的，因为对于很大一部分的实际问题他们找不到适用的有效算法；理论家们也同样沮丧，因为他们也无法找到有效的方法来证明这些问题确实是很难解决的。

人们在这个领域已经做了大量的工作，这导致一个新的机制的发展。这个机制是将新出现的这些问题在特定的技术意义上归类为“难以解决”的问题。虽然这些工作的大部分都超出了本书的范围，但是中心思想是相通的。我们在这里介绍这些工作是因为对于每个程序员来说，当他们面临一个新问题时，他们应该知道这个问题有可能是那些没人知道任何有效算法的问题，自然也无法保证能找到一个有效的解决方案。

在前面两节中，我们研究了下面两个观点，这两个观点源于 20 世纪 30 年代艾伦·图灵的开创性工作：

- 普遍性。一个图灵机能够执行任何可以用物理存在的计算设备描述的计算（判定语言或者计算函数）。这个想法被称为邱奇 - 图灵理论。把这个理论推广到自然世界时是不能被证明的（但它有可能是错的）。支持这个理论的证据是，数学家和计算机科学家已经研究出大量的计算模型，但是它们全部被证明与图灵机一致。
- 可计算性。无法通过图灵机来解决（根据普遍性，也可以说是通过任何物理存在的计算设备都无法解决）的问题是存在的。这是一个著名的数学事实。著名的停机问题（没有程序可以保证能够确定一个给定的程序是否会停机）就是这样的一个问题。

在目前的情况下，我们感兴趣的是第三个观点，即计算设备的效率问题：

- 邱奇 - 图灵理论的扩展。一个图灵机能够有效地执行任何可以用物理存在的计算设备描述的计算（判定语言或者计算函数）。

同样的，这个命题也适用于对自然世界的描述，因为所有已知的物理存在的计算设备都可以被一个图灵机模拟出来，只不过运行的成本可能会以多项式量级增加。例如，假设给定的任何算法都能在你的计算机上以与  $T(n)$  成比例的时间内完成运行（其中  $n$  是输入符号的数量），则我们可能会构造一个图灵机，它执行相同的计算所需的时间与  $T(n)^2$  成正比。反过来说，我们的程序 TuringMachine 证明了任何执行时间与  $T(n)$  成正比的图灵机都能在你的计算



机上以正比于  $T(n)$  的时间来模拟。邱奇-图灵理论的扩展意味着，从理论上来说，为了造出更高效的计算机，我们只需要关注于改进当今计算机设计的实现技术，而不是创造新的设计方案。

**概览** 难解性理论的目的是将能在多项式级别时间 (polynomial time) 内解决的问题从需要在指数级别时间 (exponential time) 内解决的问题中分离出来。我们稍后会给出这些术语的定义，但是理解难解性的第一步是真正理解指数增长的性质。

823

你可以试着估算下面的时间。这里你只需要粗略地计算，不用考虑细节。

- 地球的年龄大概为  $10^{17}$  秒。
- 地球的表面大约有  $10^{15}$  平方米。
- 一台现代超级计算机每秒大约可执行  $10^{16}$  条指令。

把这些数字相乘，你可以看到如果我们在地球的每平方英寸上有一台超级计算机，它们并行工作与地球年龄一样长的时间，我们可以计算出它们能执行  $10^{51}$  条指令。作为参考你需要知道， $10^{51}$  要比  $52!$  和  $2^{200}$  这两个数字小得多。

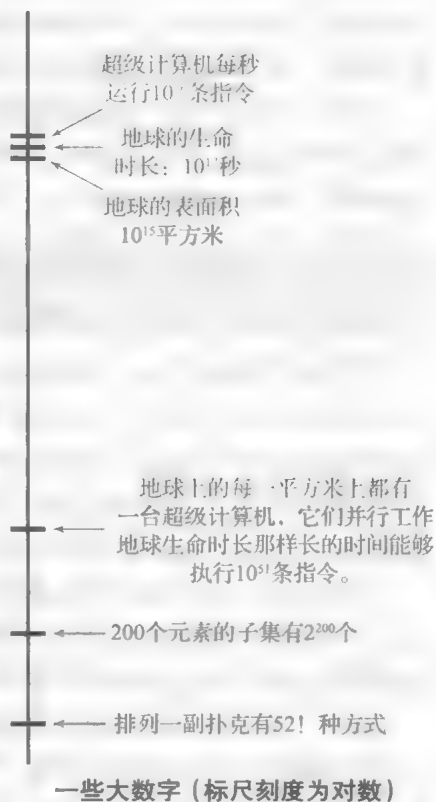
作为例子，假设你想计算一副随机洗牌的纸牌的所有可能性，你可以将这个想法忘掉了，因为一共有  $52!$  种可能性，在这个世界上没有人可以给出所有的可能性。事实上，你检查到一副牌是你想要的排列序列的概率要低于 0.000 000 000 000 005，不深入思考你无法认识到这一点。

指数级的增长远远超过技术变化的速度：一台超级计算机可能比一个算盘要快  $10^{15}$  倍，但是它们都解决不了需要计算  $2^{200}$  步的问题。另外，开发一个需要运算那么多步的算法并不困难，但显然它们即便耗尽所有的可用资源也无法获得你想要的答案，如接下来的几个例子所示。

**问题的规模。**这个理论的目的是找到针对大量算法的共性特征描述，图灵机模型使得我们可以对我们的约定进行精确的描述。我们的第一个约定是我们通过指定输入的位数 (在一个常数内) 来表示问题的规模。由于我们总是假设我们的字母表的大小是常数，“常数因子”的意思是在一个图灵机的模型中，我们将纸带上最初含有的符号数记为  $n$  (在一个 Java 程序中，我们把  $n$  作为命令行参数或者标准输入的数量)。

**最坏情况。**本节中的所有理论都建立在最坏情况分析的基础上。也就是说，对于给定大小的问题，无论输入是什么，我们都想找到关于算法的性能保障 (下限)。如果一个算法在能体现它优势的输入上很快，但是在相对较少的输入上表现很慢，那么我们认为它是慢的。采取这种悲观主义方法的原因主要有两个：

- 通常来说，将运行时间上限作为输入数据规模的一个函数的方式比较简单，而用输入数据的其他属性来表示则表示困难。例如，研究平均情况可能是一个有吸引力的选择，但是这需要为输入开发一个概率模型 (这本身就是一个挑战)，并基于这个模型



824



对算法行为进行数学分析（这可能更具有挑战性）。

- 在最坏情况下仍然能保证性能的算法是一个有价值的目标，经常在实践中被运用并且被实际应用所需要。例如，你肯定会更愿意见到用来让你的飞机着陆、控制你的汽车刹车或者控制你的起搏器的软件在最坏的情况下仍然有性能的保障。

我们之所以在一开始就强调这个问题，是因为计算理论的研究成果经常被误解。具体来说，在一些典型的实际情况下，我们不能用最坏情况的性能来预测性能。这个问题需要基于科学方法的更复杂的分析，正如我们在 4.1 节中讨论的那样。相反的，我们关注最坏情况的目的是使得我们可以更好地关注计算理论中最基本的问题。

多项式时间算法。你在本书中已经看到我们针对很多问题提出了有效的解决算法。难解性理论的第一步就是尝试将这些问题放在一类。我们这里不再讨论那些问题的细节，将问题简化为找到一个最坏情况的运行时间上限。首先，我们给出如下定义：

**定义：**多项式时间算法是运行时间取决于输入的规模  $n$ ，对于所有输入的运行时间上限为  $a \times n^b$  的算法，其中  $a$  和  $b$  都是正常数。

825 为了便于本节讨论，我们不关心常量的值，而是关注于一个算法对于所有输入的运行时间上限。例如，我们在 4.2 节中学习的排序算法就是一种多项式时间算法，我们证明了它在最坏情况下的运行时间与  $n \log n$  或者  $n^2$  成正比。如果一个问题有一个多项式时间算法，那么我们认为这个问题是“容易解决的”。我们的目标是将“容易解决的”问题从我们接下来要考虑的“难以解决的”问题中分离出来。

指数级时间算法。也有很多问题，我们不知道任何有效的算法以用作它们的解决方案。难解性理论的第二步是尝试将所有这样的问题放在一类。与前面一样，我们不讨论那些不必要的细节，将分析简化成建立一个最坏情况的运行时间下限。为此，我们给出以下定义：

**定义：**指数级时间算法是运行时间取决于输入的规模，且对于无限多的输入其运行时间下限为  $2^{a \times n^b}$  的算法，其中  $a$  和  $b$  都是正常数。

这次我们同样不关注常量的值，而是关注于对于某一类无限多的输入，指数级时间算法运行时间的下限。例如，2.3 节中的汉诺塔问题的解决方案和与它相关的算法是指数级的，因为我们证明了它们的运行时间与  $2n$  成正比。需要说明的是，这个定义将  $1.5n$ 、 $n!$  和  $2^{\sqrt{n}}$  的运行时间也认为是指数级的（见练习 5.5.3）。如果对于一个问题我们只知道指数级的算法，我们认为这个问题是“难以解决的”。

考虑到用多项式时间算法解决的问题和需要指数时间算法的问题之间存在巨大的性能差异，你可能会认为这种差异很容易区分，但事实并非如此，我们会在本节中完整地讨论难解性理论的信息，并且证明这个令人惊讶的结论。

826 **例子** 为了更好地理解这些概念，接下来我们会在涉及许多不同类型数据的算法问题中讨论这些概念。有很多是科学家、工程师和应用程序员（和你）正在解决并且渴望解决的问题，我们做的这些事为讨论这些问题提供了一个更加正式的舞台。所有这些问题都有许多重要的应用，但是我们不会对这些问题进行详细的描述，从而避免让你从这些问题最根本的属性上分散注意力。

数字。作为第一个例子，我们来讨论可以处理任意精度的整数的算法。Java 中的 `BigInteger` 类可以帮助你完成这些计算。例如，通过使用 `BigInteger`，你可以很简单地使用

小学学过的知识在二次方时间内完成两个  $n$  位整数相乘的运算（当然，现在已经有更高效的算法）。但是下面的问题似乎要困难得多：

**素数分解：**为一个给定的  $n$  位正整数找到其素数因子。

你可以通过将 Factors（程序 1.3.9）转换为 BigInteger 来解决这个问题（见练习 5.5.36）。但是这个解决方案对于多位数的  $n$  来说是不可行的，因为对于一个  $n$  位的素数来说循环迭代的次数是  $10^{n/2}$  次。例如对于 1000 位的素数，它将迭代大约 10 500 次。没有人知道解决这个问题的合适方法。事实上，互联网商业使用的 RSA 协议之所以能提供安全性就是基于这种分解的困难程度。

**子集。**作为第二个例子，我们来研究子集的求和问题，这个问题包含了我们在 4.1 节中提到的三集合的问题。

**子集的和：**为  $n$  个给定的整数找到一个（非空的）子集，这个子集的和为 0 或者返回“这个子集不存在”。

原则上来说，你可以通过编写一个尝试所有可能性的程序来解决这个问题（见练习 5.5.4），但是对于一个很大的  $n$ ，程序将不会结束，因为对于  $n$  个元素的集合来说有  $2^n$  个不同的子集。有一个很简单的方法可以让你相信这个事实，将一个任意  $n$  位的二进制数对应于  $n$  个元素的子集，根据 1 的位置来决定子集中包含的元素。右边给出了一个例子，每个二进制数字的右边是与之对应的子集的和。同样的，当  $n = 200$  时考虑所有的可能性是不可能的。没有人能知道在  $n$  很大的时候能保证解决这个问题的算法（即使我们可以解决一个特定的问题实例——例如，我们在寻找的早期就找到了一个和为 0 的子集）。

在第三个例子中，为了强调子集的概念会出现在各种各样的问题中，我们考虑以下问题：

**顶点覆盖：**给定一个图  $G$  和一个整数  $m$ ，在  $G$  中寻找一个最多有  $m$  个顶点的子集，这些顶点应该与所有的边相连接，或者返回“这样的子集不存在”。

右图给出了一个例子，大小为 1 和 2 的子集都用深色的顶点标注出来，有两种情况可以作为这个问题的正确答案（即没有任何一条边的两个顶点都是灰色的）。为了对应用程序的功能有个更加自然的感受，我们可以将顶点看作路由器，将边看作网络连接。然后顶点覆盖就可以告诉一个网络攻击者是否可以通过瘫痪  $m$  个顶点达到令所有通信失效的目的（或者告诉防御者是否可以通过保护  $m$  个顶点保证至少有一条通信链路受到保护）。在这个例子中，当  $m$  较小的情况下，正确答案的数量是关于  $m$  的多项式，而  $m$  较大的时候则是指数形式（见练习 5.5.6）。很快我们会再一次研究一个通过尝试所有可能性来解决这个问题的程序，但是我们要强调的是只有  $m$  很小的时候才能发挥作用，当  $m$  很大时，这个程序不会终止。

问题：

为给定的 5 个整数  
找到和为 0 的子集

1 342, -1 991, 231, -351, 1 000

所有可能性：

|           |        |
|-----------|--------|
| 0 0 0 0 1 | 1 000  |
| 0 0 0 1 0 | -351   |
| 0 0 0 1 1 | 649    |
| 0 0 1 0 0 | 231    |
| 0 0 1 0 1 | 1 231  |
| 0 0 1 1 0 | -120   |
| 0 0 1 1 1 | 880    |
| 0 1 0 0 0 | -1 991 |
| 0 1 0 0 1 | -991   |
| 0 1 0 1 0 | -2 341 |
| 0 1 0 1 1 | -1 341 |
| 0 1 1 0 0 | -1 760 |
| 0 1 1 0 1 | -760   |
| 0 1 1 1 0 | -2 111 |
| 0 1 1 1 1 | -1 111 |
| 1 0 0 0 0 | 1 342  |
| 1 0 0 0 1 | 2 342  |
| 1 0 0 1 0 | 991    |
| 1 0 0 1 1 | 1 991  |
| 1 0 1 0 0 | 1 673  |
| 1 0 1 0 1 | 2 673  |
| 1 0 1 1 0 | 1 322  |
| 1 0 1 1 1 | 2 322  |
| 1 1 0 0 0 | -649   |
| 1 1 0 0 1 | 351    |
| 1 1 0 1 0 | -1 000 |
| 1 1 0 1 1 | 0      |
| 1 1 1 0 0 | -418   |
| 1 1 1 0 1 | 682    |
| 1 1 1 1 0 | -769   |
| 1 1 1 1 1 | 231    |

解决方案：

1342, -1991, -351, 1000

子集和问题的一个示例

827

“难以解决”的问题。对于刚刚描述的所有问题，要想找到解决方案都需要“尝试所有的可能性”。每个用计算机解决问题的人都必须明白，这样的解决方法并不总是有效的，因为会面临指数增长的问题。这与你凭直觉认为计算机速度很快，如果有足够的时间就可以解决任何问题的看法是想反的（你可以问问你的朋友，看他们是否觉得计算机已经快到足以算出一副扑克牌的所有排列）。如果一个问题的已知算法都要用指数级时间解决，那么我们认为这个问题是“难以解决”的，就像前文定义的一样，我们通常假定一个指数级时间算法的大小限定在 1000 以内。因为没有人可以等待一个算法执行  $2^{1000}$  或者  $1000!$  步，不管计算机的运行速度有多快。

“易于解决”的问题。相比之下，我们面临的很多问题并不需要我们尝试所有的可能性（甚至是大部分的可能性），并且即使这些问题的规模很大，我们也可以设计出有效解决的算法。如果对于一个问题我们知道一种多项式时间算法可以解决它，那么我们认为这个问题是“易于解决”的，就像前文定义的一样。一般来说，我们常用到的基本计算功能都是建立在这样的算法上的。

一条界线。有时候，“简单”问题和“困难”问题之间的界线是很好判别的。例如：

最短路径。在一个给定的图中，给定一个顶点  $s$  和  $t$ ，找到一条从  $s$  到  $t$  最多有  $m$  条边的路径，或者报告不存在这样的路径。

我们在 4.1 节中的程序 PathFinder 给出这个问题的最优解（找到最短路径），并给出了一个即时的解决方案。但是我们没有研究过下面这个问题的算法，这个问题看起来与最短路径似乎是一样的。

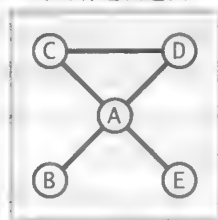
最长路径：在一个给定的图中，给定一个顶点  $s$  和  $t$ ，找到一条从  $s$  到  $t$  最少有  $m$  条边的路径，或者报告不存在这样的路径。

问题的麻烦之处就在于，就我们所知，这两个问题的难度几乎处于两个极端。广度优先搜索为第一个问题提供了一个线性时间的解决方案，但是对于第二个问题，所有的已知算法在最坏的情况下都需要指数时间来完成，因为它们有可能需要检查所有的路径。

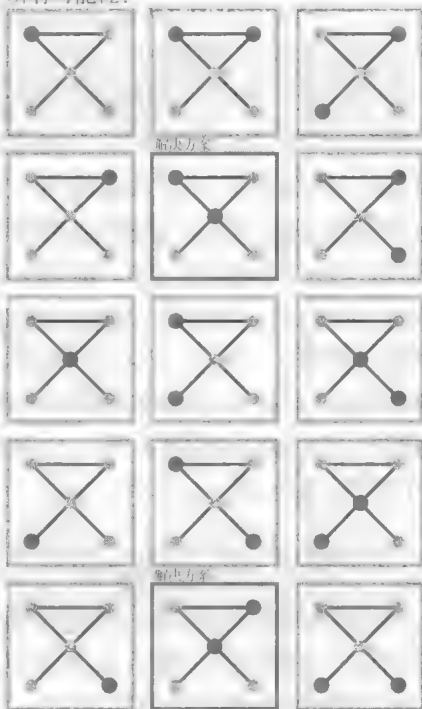
一般来说，当我们知道一个问题是“易于解决”的时候，我们就可以改进算法，并且希望通过技术上的提升让我们能够用这个算法解决越来越多的问题实例（实际上，典型的“易于解决”的问题是可以保证在一个运行时间界限内解决的，这个界限由输入规模的多项式来决定，如  $n^2$  或  $n^3$ ）。当我们知道一个问题是“难以解决”的，我们就不能指望技术上的进步来帮助我们解决问题。那么哪些问题是“易于解决”的，哪些问题是“难以解决”的呢？我们即将考虑的理论能够帮助我们回答这个问题。从实际的角度看，这一理论与可计算性理论一样重要。

问题：

找到小于或等于 2 个的一组顶点，且这些顶点与所有边相连接



所有可能性：



一个顶点覆盖实例

**可满足性** 有四个特殊的问题被称为可满足性问题，这些问题在我们讨论难解性时很重要。

**线性方程的可满足性：**给定一组包含  $n$  个变量的  $n$  个线性方程，对所有变量找到一个有理数解来满足所有方程，或者报告没有解存在。

**线性不等式的可满足性：**给定一组包含  $n$  个变量的  $m$  个线性不等式，对所有变量找到一个有理数解来满足所有不等式，或者报告没有解存在。

**整数线性不等式的可满足性：**给定一组包含  $n$  个变量的  $m$  个线性不等式，对所有变量找到一个整数解来满足所有不等式，或者报告没有解存在。

**布尔可满足性：**给定一组包含  $n$  个布尔变量的  $m$  个方程，对所有变量找到一个布尔值解来满足所有方程，或者报告没有解存在。

由于计算机在工业和商业中得到了广泛的应用，这些问题是广泛适用的，并且在过去的几十年中发挥了核心作用。尽管这些问题有相似之处，但是要解决这些问题所面临的挑战在本质上有着惊人的差异，下面我们对它们进行简要描述。

**线性方程的可满足性。**你熟悉的这个问题是“联立解方程式”，解决这个问题的算法一般被称为高斯消元法 (Gaussian elimination)，这个算法可以追溯到中国古代，并且自 18 世纪以来一直在代数类中教学。这种基本的算法很容易实现 (见练习 5.5.2)，并且可以在标准的数值型线性代数库中使用。当变量的值是 `double` 型时，实现一个对于所有的输入都能正常工作的高斯消元程序实际上是一个挑战 (因此建议使用函数库中实现好的版本)。并不是所有版本的高斯消元法都是多项式时间的，因为中间计算有可能按照指数规律进行，但是有一些标准版本的高斯消元法已经被证明是多项式时间的 (使用库函数的另一个理由)。尽管存在这些技术挑战，但是认为这个问题“易于解决”是合理的。

问题:

$$\begin{aligned} 4x - 2y - z &= -1 \\ 4x + 4y + 10z &= 9 \\ 12x + 4y + 8z &= 21 \end{aligned}$$

解:

$$x = 5/4, y = 7/2, z = -1$$

一个线性方程可满足性的实例

830

**线性不等式的可满足性。**现在假设我们联立的方程中允许存在不等式，而不仅仅只有等式方程 (而且我们可以允许比变量更多的不等式)。这个变化产生了一个经典问题的一个版本，即线性规划 (linear programming, LP) 问题。这个问题是 20 世纪中期为了战争时的物流规划而提出的，有一种著名的解决这个问题的算法叫单纯形法 (simplex method)，这个算法是乔治·丹齐格在 1947 年发明的，并一直成功使用至今。单纯形法比高斯消元法复杂得多，但是经过一些研究你可以理解这个方法背后的思想，而且这种算法也是可以被广泛实现的。但是单纯形法可能需要指数级时间来运行 (或者更糟)。关于线性规划是否存在多项式时间算法的问题讨论了数十年，直到 20 世纪 80 年代中期才得以解决。从技术上讲，现在我们认为这个问题是“易于解决”的。线性规划的现代实现方法被广泛用于各种工业和管理应用中。航空时刻表或者快递邮件可能都是线性规划问题解决方案的一部分，其中可能涉及数十万个不等式。

问题:

$$\begin{aligned} 6x - 10y - z &\leq 0 \\ 2x + 2y + 5z &\leq 76 \\ 3x + y + 2z &\leq 54 \\ x, y, z &\geq 0 \end{aligned}$$

解:

$$x = 10, y = 51/10, z = 9$$

一个线性不等式可满足性的实例  
(寻找线性规划的表达)

**整数线性不等式的可满足性。**如果线性规划问题中的变量代表航空公司的飞行员或者卡车，则有必要保证解决方案中变量的值是整数。这个问题被称为整数线性规划 (Integer Linear Programming, ILP)。在某些应用程序中，我们将变量的值限制为 0 或者 1，这种版本

被称为 0/1ILP。你可能会惊讶地发现这些问题没有多项式时间的算法。解决 ILP 的一个方法是先解决相同的 LP 以得到一个分数解，然后将这个分数解转化为最接近的整数——这可能是一个解决方案的起点，但它并不总是有效的。有一些软件包可以解决实际中出现的 ILP 实例：这种软件被广泛用于各种工业和商业应用中。但是存在这种软件的速度太慢而不能使用的实例，所以研究人员仍然在寻找更快的算法。据我们所知，LP 和 ILP 之间的区别代表着我们可以在多项式时间内解决的问题与需要指数时间解决的问题之间的一条“细线”。ILP 是“易于解决”还是“困难”的？这个研究问题已经研究了几十年，目前还没有找到它的答案。

问题：

$$\begin{aligned}x' + z &= \text{true} \\x + y' + z &= \text{true} \\x + y &= \text{true} \\x' + y' &= \text{true}\end{aligned}$$

简单表示：

$$s = (x' + z)(x + y' + z)(x + y)(x' + y') = \text{true}$$

所有可能性 (T 代表真, F 代表假)：

| $x$ | $x'$ | $z$ | $(x' + z)$ | $(x + y' + z)$ | $(x + y)$ | $(x' + y')$ | $s$ |
|-----|------|-----|------------|----------------|-----------|-------------|-----|
| F   | F    | F   | T          | T              | F         | T           | F   |
| F   | F    | T   | T          | T              | F         | T           | F   |
| F   | T    | F   | T          | F              | T         | T           | F   |
| F   | T    | T   | T          | T              | T         | T           | T   |
| T   | F    | F   | F          | T              | T         | T           | F   |
| T   | F    | T   | T          | T              | T         | T           | T   |
| T   | T    | F   | F          | T              | T         | F           | F   |
| T   | T    | T   | T          | T              | T         | F           | F   |

解：

$$\begin{aligned}x &= \text{false} & x &= \text{true} \\y &= \text{true} & y &= \text{false} \\z &= \text{true} & z &= \text{true}\end{aligned}$$

一个布尔可满足性的实例 (SAT)

问题：

$$\begin{aligned}7x - 10y - z &\leq 1 \\2x + 2y + 5z &\leq 77 \\3x + y + 2z &\leq 54 \\x, y, z &\geq 0\end{aligned}$$

解：

$$x = 10, y = 6, z = 9$$

一个整数线性不等式可满足性的实例  
(寻找整数线性规划的表达式)

布尔可满足性。如果我们联立的方程组是布尔方程组，那么就有了布尔可满足性的问题 (SAT)。如果你不熟悉布尔代数或者需要复习，请阅读我们在 7.1 节开头进行的处理。我们的变量有两个值——假和真，我们使用三个操作： $x'$  表示对  $x$  取非，即当  $x$  为真的时候  $x'$  为假，当  $x$  为假的时候  $x'$  为真； $x + y$  表示“ $x$  或  $y$ ”，即当  $x$  和  $y$  都为假时， $x + y$  为假，其他情况都为真； $xy$  表示“ $x$  与  $y$ ”，即当  $x$  和  $y$  都为真时， $xy$  为真，其他情况都为假。左边是一个例子。通常情况下，我们假设等号右侧全部为真并且等号左侧的每个方程式都没有使用与操作 (见练习 5.5.8)。我们也可以使用一个简单表示：将所有方程的等号左侧用与操作连接起来放在单个方程式中。左边还显示了一个表格，这个表格列出了每个变量和每个左侧表达式所有可能的值，表格中还标出了哪些是我们需要的解决方案 (即所有值都为真的两行)。

SAT 看起来似乎是一个抽象的数学问题，但是它有许多重要的应用。例如它可以模拟电路的行为，所以它在现代计算机的设计中起着至关重要的作用。与 ILP 一样，人们已经开发出了可以解决实际中出现的问题实例的算法，并且“SAT 求解器”已经被广泛使用。唐纳德·E. 克努特 (D. E. Knuth) 估计一个可以适应工业强度的 SAT 求解器是一个价值十亿美元的产业。但是 SAT 与 ILP 一样，在已知的所有算法中没有多项式时间的算法——每个 SAT 求解器都会在一些实例问题上以指数时间来运行。

可满足性问题是各类计算型应用程序的典型示例。我们很容易用公式表示问题，也很容易找到各种各样的应用程序，但是辨别“易于解决的”问题和“难以解决的”问题之间的差异可能是非常具有挑战性的。这个挑战在计算发展的初期就已经显现出来，而且它还促进了我们将要描述的理论框架的发展。

**搜索问题** 在“易于解决的”问题和“难以解决的”问题之间存在巨大的差距，这使得我们可以用一个简单且正式的模型来划分它们之间的边界。我们的第一步是描述研究的问题的类型：

**定义：**搜索问题是指这样一类问题，它们可以在多项式时间内检查一个给定的解决方案是否能解决一个给定的问题实例。对于一个搜索问题，如果存在一个算法对于任意一组给定的输入都能给出一个解决方案或者报告不存在解决方案，那么我们认为这个算法解决了这个搜索问题。

到目前为止，本节提到的所有问题（排序、分解乘法、最短路径、最长路径、布尔可满足性等）都是搜索问题。要确定一个问题是不是搜索问题，我们只需要确定对于任意一个解决方案，我们都可以高效地验证它是否正确。解决一个搜索问题就像大海捞针，唯一的条件是当你看到针的时候你可以认出这是一根针。例如在布尔可满足性问题中，如果你给每个变量赋值，那么你可以轻松地检查每个方程或者不等式是否被满足，但是搜索一个这样的赋值（或者确定是否存在）却是一个艰巨的任务。

**NP** 通常用于描述搜索问题。许多人把 **NP** 当作是“非多项式”（not polynomial）的缩写，但是事实并非如此——我们将在本节稍后的部分描述这个命名的含义。

**定义：****NP** 是所有搜索问题的集合。

**NP** 是对科学家、工程师和应用程序员都渴望能够在合理时间内解决的所有问题的精确描述。在后面，我们对 **NP** 中那些已经讨论过的问题进行了总结。接下来我们更加仔细地研究几个例子。

833

**NP 中的子集求和问题。**想要证明一个问题是搜索问题，只要提供的 Java 代码能（在多项式时间内）检查一个推定的解决方案是否在给定的输入下真正地解决了这个问题。例如，假设我们将子集求和问题的输入保存在一个整数数组 `values[]` 中，然后维护一个布尔型数组 `inSubset[]`，如果这个数组中标为 `true`，则表示子集中包含相应的值。在上面的表格里，我们给出了前文“子集求和”示例的值。通过这种表示，检查子集求和是否为 0 的 Java 代码变得十分简单：

| i | values[i] | inSubset[i] |
|---|-----------|-------------|
| 0 | 1342      | true        |
| 1 | -1991     | true        |
| 2 | 231       | false       |
| 3 | -351      | true        |
| 4 | 1000      | true        |

```
public static boolean check(int[] values, boolean[] inSubset)
{
 int sum = 0;
 for (int i = 0; i < n; i++)
 if (inSubset[i])
 sum += values[i];
 return sum == 0;
}
```

在这种情况下，检查解决方案是否有效的操作在对数据进行扫描后即可完成，这个操作只需要线性时间。这是 **NP** 中的典型问题之一，其表达的基本特点是：当我们看到 **NP** 问题的一个解决方法时，我们可以轻易地辨认出它是不是正确答案。

**NP 中的顶点覆盖问题。**我们可以用同样的机制来证明顶点覆盖问题在 **NP** 中。假设我们的输入用图 *G* 表示，这个图采用 4.1 节中使用的图数据类型来存储，并且用一个整数 *m*



来表示子集大小的上限。我们可以用一个布尔类型的数组 `inSubset[]` 来表示顶点覆盖问题的解决方案，数组中的每个 `true` 元素对应每个被包含在解决方案中的顶点。然后，为了检查这个顶点的子集是否符合顶点覆盖的定义，我们需要：

- 确保子集中的顶点数量 (`inSubset[]` 中 `true` 元素的个数) 小于或者等于  $m$ 。
- 检查图中所有的边，确保没有边连接的两个顶点都不在子集中。

834

这些检查很容易完成 (见练习 5.5.7)。同时，这些检查的成本与输入的大小是呈线性关系的。

NP 中的 0/1 ILP。现在我们来分析 0/1 ILP 是否在 NP 中。假设我们的输入由一个  $m \times n$  的矩阵 `a[][]` 表示，等号右侧是一个长度为  $m$  的向量 `b[]`。我们用长度为  $n$  的向量 `x[]` 来表示一个解。然后，为了检查一个声明的解向量 `x[]` 是否确实是一个解，我们需要：

- 检查每个元素 `x[j]` 是 0 还是 1。
- 检查每个不等式  $i$  是否都有

$$a[i][0] * x[0] + a[i][1] * x[1] + \cdots + a[i][n-1] * x[n-1] \leq b[i]$$

这些检查很容易在与输入的大小呈线性关系的时间内完成 (见练习 5.5.11)。相似地，我们可以得出 SAT 是一个 NP 问题。但是要证明 ILP 是 NP 问题，我们需要更仔细地论证 (超出我们的范围)，因为 ILP 问题 (没有 0/1 限制) 的解决方案中的值从理论上来说可能是指数量级的。

找到一个解决方案。要解决一个搜索问题，真正的计算负担在于如何找到一个解决方案。正如指出的那样，我们知道解决许多问题的最好方法基本上是尝试所有可能的解决方案。在本节稍后的部分我们会考虑一个 SAT 的例子 (见程序 5.5.1)。本质上来说，程序需要做的就是对于大小为  $n$  的问题找出全部的  $2^n$  个子集，这需要指数时间来完成。

我们通过搜索问题来定义 NP 的方法只是文献中被广泛接受的三种方法之一，这些文献描述了一些问题，这些问题构成了难解性研究的基础。另外的两种方法是决策问题 (是否存在解决方案?) 和优化问题 (这是最好的解决方案吗)? 例如有三种提出顶点覆盖问题的方式。给定图  $G$ ，我们有下列三个问题：

- 优化：对图  $G$  找到一个包含顶点数量最少的顶点覆盖解决方案。
- 决策：当最多有  $m$  个顶点时，图  $G$  是否存在顶点覆盖的解决方案?
- 搜索：为图  $G$  找到一个最多有  $m$  个顶点的顶点覆盖解决方案。

虽然在技术上并不相同，但这三种定义 NP 的方法之间的关系已经得到了深入的研究，我们得出的主要结论也都适用于所有这三类问题。为了避免混淆，我们专注于研究搜索问题。当我们使用“LP”这个名字时，我们的意思是“LP 的搜索问题表述”。从这个角度来看，我们在使用“在 NP 中”和“搜索问题”这两个术语时是不带注解地交换使用的，所以你应该小心，不要认为这两个术语是完全相同的。

835

不确定性。NP 中的 N 代表不确定性 (nondeterminism)。它代表了这样的观点：扩展计算机力量的一种方式 (从理论上讲) 是赋予其不确定性的力量，即当一个算法面临在几个选项中做出选择时，它有能力“猜出”正确的那个。不确定性可能是一个数学上的虚幻构想，但它是一个很有用的想法 (例如，在 5.1 节中我们可以看到不确定性对于算法设计是很有用的——通过模拟一个非确定性机器可以有效地解决正则表达式识别问题)。

对于任何人来说都需要先对一个解决方案进行检查才能证明这个方案是有效的。就我们的讨论而言，我们可以将一个不确定性机器的算法看作“猜测”问题的解决方案，然后验证这个解决方案是否是正确的。在图灵机中不确定性的表示非常简单，就像为一个给定的状态和一个给定的输入定义两个不同的后续状态，然后将解决方案表示为能到达期望结果的所有合法路径。



例如，我们可以构建一个不确定的图灵机，这个图灵机通过将右侧展示的机器附加到一个 SAT 检查器的头上来解决 SAT 问题，并且运行的时候还要执行纸带初始化，这是为了将 SAT 实例放到最左端的纸带头上。这个机器的不确定部分用来简单地猜测每个变量的正确值，然后把解决方案放在纸带上。随后一个（确定的）像练习 5.5.38 中的 SAT 检查器可以验证这个答案。如果你觉得机器猜测这个想法不好，那么你可以这样思考不确定性：一个不确定的机器当且仅当存在某条路径可以让机器从开始状态到一个 Yes 状态时，才会接受输入字符串。在这种情况下答案是明确的：若公式成立，则有一条通过不确定部分的路径可以将满足题目要求的值写在纸带上，一条通过确定部分的路径可以到达 Yes 状态。

假设一台机器能够猜出答案看起来似乎是一种奢望，但是如果我们将一个不确定的图灵机转换为一个确定的机器，那么这个想法就会变得更加可行了，就像我们在前文对 NFA 做的那样。最终我们得到的结果是一个尝试检查所有  $2^n$  个可能输入值的确定的图灵机。任何在 NP 中的问题都可以在指数时间内解决（通过相同的推理）。问题的关键不在于找到答案的可能性，而是在于成本。那么存在一个能在多项式时间内找到并检查解决方案的确定的图灵机吗？

下面的定义与我们介绍的定义等价：“NP 是所有能在不确定图灵机上在多项式时间内解决（并检查）的问题的集合。”你接下来就会看到这种表达是让我们能够证明关于 NP 的其他特性的一个必要步骤。

“易于解决”的搜索问题。NP 的定义中没有说明找到解决方案的困难之处；它仅仅描述了如何检查提出的解决方案是否是有效的。构成难解性研究基础的两组问题中的第二个问题，我们称之为 P 问题，它与寻找解决方案的难度有关。

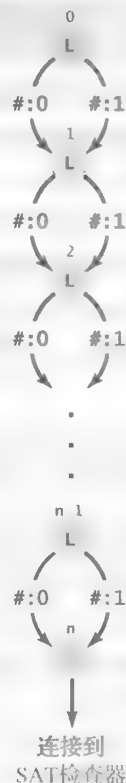
**定义：**P 问题是所有可以在多项式时间内解决的搜索问题的集合。

对于在 P 中的问题，必定存在一个多项式时间算法以解决这个问题。从数学的角度上来说，“算法”的意思是“确定的图灵机”，“多项式时间”的意思是“上限是输入纸带上输入位数的一个多项式函数”。“多项式”这个词并没有指定任何具体细节——我们仅仅是用它来区别指数时间，如指数时间算法定义所述。

对于某一个具体问题，如果我们能写出一个在多项式时间内解决问题的 Java 程序，那么我们认为这个问题在 P 中。排序问题属于 P，因为插入排序的运行时间与  $n^2$  成正比（我们这里并不关心是否有更快的排序算法），所以说像最短路径算法、线性方程求解和很多其他算法都是属于 P 的。对数算法、线性算法、二次算法和三次算法都属于多项式时间算法，所以这个定义肯定覆盖了目前我们研究的经典算法。

对于一个搜索问题，找到一个有效算法来解决它就证明了它属于 P。换句话说，P 只不过是对于一类问题的一个精确描述，这类问题包括所有能够被科学家、工程师和程序员用程序在一个合适的时间内完成的问题。在后文，我们列出了以前讨论过的在 P 中的问题。

“难以解决”的搜索问题。如果一个搜索问题不在 P 中，我们便知道没有多项式时间算



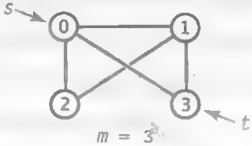
解决 SAT 问题的不确定图灵机

法能解决这个问题。我们使用“难以解决”(intractable)这个词来形容这样的问题。

**定义：**一个问题若不存在多项式时间算法的解法，那么它就是难以解决的。

如果一个问题难以解决，我们便不能保证可以在一个合适的时间内解决这个问题，除非输入的规模很小。在本节中，我们将会遇到几个被认为是难以解决的著名搜索问题。

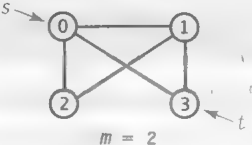
837

| 问题        | 输入                           | 描述                                         | 多项式时间算法 | 实例                                                                                            | 解决方案                                                         |
|-----------|------------------------------|--------------------------------------------|---------|-----------------------------------------------------------------------------------------------|--------------------------------------------------------------|
| 最长路径      | 图 $G$<br>顶点 $s, t$<br>整数 $m$ | 在图 $G$ 中找到一个从 $s$ 到 $t$ 、长度大于或等于 $m$ 的简单路径 | ?       | <br>$m = 3$ | 0-2-1-3                                                      |
| 质因子分解     | 整数 $x$                       | 找到 $x$ 的一个非平凡因子 (质因子)                      | ?       | 97605257271                                                                                   | 8784561                                                      |
| 子集求和      | 整数集合                         | 找到一个和为 0 的子集                               | ?       | 32 3 -44 8 12                                                                                 | 32<br>-44<br>12                                              |
| 整数线性不等式求解 | $n$ 个变量<br>$m$ 个不等式          | 对变量分配整数值，使得所有不等式被满足                        | ?       | $y - x \leq 1$<br>$5x - z \leq 2$<br>$x + y \geq 2$<br>$z \geq 0$                             | $x = 1$<br>$y = 2$<br>$z = 3$                                |
| 布尔方程组求解   | $n$ 个变量<br>$m$ 个方程           | 对变量分配 true/false 值，使得所有方程被满足               | ?       | $x + y = \text{true}$<br>$y + z' = \text{true}$<br>$x' + y' + z' = \text{true}$               | $x = \text{true}$<br>$y = \text{true}$<br>$z = \text{false}$ |

这些问题都属于 P——见下面的表格

838

属于 NP 的问题示例

| 问题      | 输入                           | 描述                                       | 多项式时间算法 | 实例                                                                                              | 解决方案                            |
|---------|------------------------------|------------------------------------------|---------|-------------------------------------------------------------------------------------------------|---------------------------------|
| 最短路径    | 图 $G$<br>顶点 $s, t$<br>整数 $m$ | 在图 $G$ 中找到一个从 $s$ 到 $t$ 、长度小于或等于 $m$ 的路径 | BFS     | <br>$m = 2$ | 0-3                             |
| 乘法      | 两个整数                         | 计算它们的乘积                                  | 学校中教的乘法 | 8784561 123123                                                                                  | 97605257271                     |
| 排序      | 可比较的值组成的数组 $a$               | 找到一个从小到大的排序并放进 $a$ 中                     | 归并排序    | bc zy mn ab                                                                                     | 3 0 2 1                         |
| 线性方程求解  | $n$ 个变量<br>$n$ 个方程           | 对每个变量分配值，使得所有方程被满足                       | 高斯消元法   | $6x + y = 4$<br>$2x - y = 0$                                                                    | $x = 1/2$<br>$y = 1$            |
| 线性不等式求解 | $n$ 个变量<br>$m$ 个不等式          | 对每个变量分配值，使得所有不等式被满足                      | 椭圆算法    | $4x - 4y \leq 3$<br>$2x - z \leq 0$<br>$2x + 2y \geq 7$<br>$z \geq 4$                           | $x = 2$<br>$y = 3/2$<br>$z = 4$ |

839

属于 P 的问题实例

想象一下，20 世纪中叶的应用数学家（以及当时为数不多的计算机科学家）制定了这些问题类别，他们希望通过这种方式来找到一种识别难以解决的问题的方法，从而避免研究这些问题，就像我们对无法解决的问题所做的那样。但是他们不知道，五十年过去了，我们仍然不知道是否能够找到一个多项式时间的解法。

**主要问题** 不确定性看起来似乎是一种幻想，它并不是我们在现实世界中能拥有的东西，认真考虑它似乎是一件荒谬的事情。虽然它能让难题变得微不足道，但是我们要不要花时间考虑一个虚无缥缈的工具呢？问题的答案是，虽然不确定性看起来似乎很强大，但是没有一个人能证明对于一个特定的搜索问题，一个不确定的机器能比确定的机器要快！换句话说，我们当然想知道哪些搜索问题在  $P$  中，哪些是难以处理的，但是没有人找到一个可以被证明是在  $NP$  中并且不在  $P$  中的问题（甚至不能证明这个问题的存在），所以我们不禁思考下面这个基本问题：

### $P = NP$ 吗？

S. Cook 在 1971 年以这种形式精确地提出了这个问题，尽管早些时候有几个研究者提出了一些近似的概念，其中包括由库尔特·哥德尔在 1956 年写给冯·诺依曼的著名的“丢失的信件”（还有 1955 年约翰·纳什给国家安全局（NSA）的信件，这封信在 2012 年被解密）。这个问题从那以后就彻底难倒了数学家和计算机科学家。这个问题的其他提问方式揭示了它的基本性质：

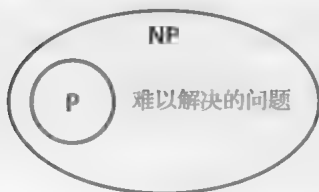
- 是否存在一些难以解决的搜索问题？
- 对于一些搜索问题，穷举法真的是最好的方法吗？
- 寻找解决搜索问题的方法比检查解决方案是否有效更难吗？
- 对于解决一些搜索问题，在不确定的计算设备上一定比在确定的计算设备上更有效吗？

无法知道这些问题的答案是一件令人沮丧的事情，因为有许多重要的实际问题属于  $NP$ ，但是可能属于也可能不属于  $P$ （已知的最优求解算法是指数级的）。如果我们可以证明一个搜索问题是难以解决的（不属于  $P$ ），那么我们可以放弃寻找一个有效的解决方案。在没有任何这样的证明的情况下，有可能是某个有效的算法未被我们发现。事实上，考虑到我们当前的知识水平， $NP$  中的每一个问题都有可能有一些有效的算法，这意味着有很多有效的算法都未被发现。我们生活在下面这两种可能中的一个：

840

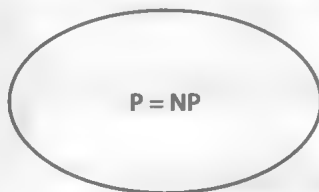
$P \neq NP$

有一些搜索问题是难以解决的。  
对于某些搜索问题，穷举法解决方案可能就是最优解。  
寻找解决搜索问题的方法比检查一个解决方案是否有效更难。  
不确定性能帮助我们更有效地解决搜索问题。



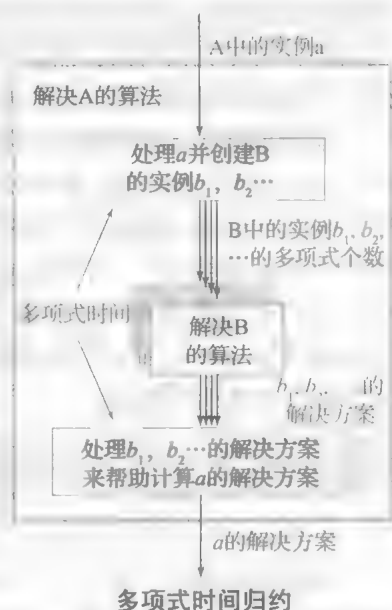
$P = NP$

所有的搜索问题都是可以解决的。  
对于 ILP、SAT、分解问题等都存在有效算法，并且所有问题都属于  $NP$ 。  
寻找搜索问题的解决方案和检查解决方案是否有效的难易程度是一样的。  
不确定性不会对我们解决搜索问题产生帮助。



但是我们并不知道哪个才是对的。很少人会相信  $P = NP$ ，而且人们已经付出了相当多的努力来证明  $P \neq NP$ ，但是这个问题仍然是计算机理论中突出的开放性研究问题。

**多项式时间归约** 进一步了解  $P = NP$ ？问题的关键是关于多项式时间归约 (polynomial time reduction)。回顾一下 5.4 节，我们说如果可以通过执行一些标准的计算步骤，加上调用一些用于解决问题 B 的子程序来解决任意一个问题 A 的实例，那么我们就可以将问题 A 归约成问题 B。在本节内容中，我们限制了子程序的调用次数并将子程序之外的执行时间限制在输入大小的多项式时间范围内。因此，如果问题 A 能在多项式时间内归约为问题 B (并且我们可以在多项式时间内求解 B)，那么我们就能够在多项式时间内求解 A。这种简化被称为库克归约 (Cook reduction)，而卡尔普归约 (Karp reduction) 则更为严格。但是，这两个我们得出的一般结论对于这两种定义都是成立的。



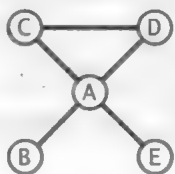
**定义：**如果问题 A 存在一种算法，它对子程序 B 的调用是多项式次数，及子程序外花费的时间是多项式的，那么我们就称问题 A 可以在多项式时间内归约为问题 B。

我们所有的证明都只使用 B 中的一个实例，如 5.4 节所述，但是理论上来说我们可以使用多项式数量个实例，只要将所有对 A 的输入转化为对 B 的输入，以及将 B 的解决方案转化为 A 的解决方案 (和所有其他的东西) 仍然是在多项式时间界限内即可。

如果问题 A 能在多项式时间内归约成问题 B，那么就可以通过 B 的一个多项式时间算法产生一个 A 的多项式时间算法。这是从软件开发得出来的一个令人熟悉的概念：当你调用一个库函数来解决问题的时候，你就是在把你的问题归约成一个调用库函数能解决的问题，而这通常是多项式时间的。

**顶点覆盖问题**

找到包含所有边且顶点数小于或等于 2 的顶点



0/1 ILP 的表示

找到满足不等式的  $x_A, \dots, x_E$  的 0/1 值

$$\begin{aligned} x_A + x_B &\geq 1 \\ x_A + x_C &\geq 1 \\ x_A + x_D &\geq 1 \\ x_A + x_E &\geq 1 \\ x_C + x_D &\geq 1 \end{aligned}$$

0/1 ILP 的解决方案

$$\begin{aligned} x_A &= 1 & x_A &= 1 \\ x_B &= 0 & x_B &= 0 \\ x_C &= 1 & x_C &= 0 \\ x_D &= 0 & x_D &= 1 \\ x_E &= 0 & x_E &= 0 \end{aligned}$$

顶点覆盖问题的解决方案

{A, C}    {A, D}

将顶点覆盖问题归约为 0/1 ILP

当你调用一个库函数来解决问题的时候，你就是在把你的问题归约成一个调用库函数能解决的问题，而这通常是多项式时间的。有一些问题很重要，因为它们证明了很多多项式时间归约是可行的，即使是一些看起来并不相似的问题。下面是一个例子。

**命题 C：**顶点覆盖问题可以归约为 0/1 ILP。

**证明：**给定一个顶点覆盖的实例，定义一组不等式，其中每个顶点对应一个 0/1 变量，每条边对应一个不等式，如左边的例子所示。值为 1 的变量对应被覆盖的顶点——满足所有与边对应的不等式的唯一方法是将值 1 分配给每个不等式两个变量中的至少一个。为了解决一个顶点覆盖问题的实例，首先求解一个 0/1 ILP 问题的实例，然后通过把每个整数变量值为 1 的顶点放入顶点覆盖问题当中来将 0/1 ILP 的解决方案转化为顶点覆盖问题的解决方案。

一般来说,可满足性问题允许进行许多这样的归约。这就是为什么这些问题的求解器被广泛地用于实际中,以及为什么它们是否属于  $P$  这个问题如此重要。这个归约并没有带给我们一个能解决顶点覆盖问题的多项式时间算法,但是它建立了这些问题之间的关系。这些关系是难解性理论的基础。

842

为了检查一下你是不是真的理解了怎么使用归约,请你花时间说服自己下面三个事实是真的:

- 如果  $A$  能在多项式时间内归约成  $B$  且  $B$  属于  $P$ , 则  $A$  也属于  $P$ 。
- 任何属于  $P$  的问题都可以归约成其他任何问题。
- 多项式时间的归约具有传递性: 如果  $A$  能在多项式时间内归约成  $B$ , 且  $B$  能在多项式时间内归约成  $C$ , 则  $A$  能在多项式时间内归约成  $C$ 。

第一个问题很容易通过统计解决  $A$  的成本来证明。第二个问题是一种空虚的真(无法找到这样的实例)。例如,排序问题可以归约成停机问题;因此,如果我们可以多项式时间内解决停机问题,那么我们就可以多项式时间内解决排序问题。虽然我们已经在多项式时间内进行排序了,但是这与我们是能否解决停机问题是完全无关的。我们将归约的传递性的证明留作练习(见练习 5.5.30)。

**NP 完全性** 我们已经知道有很多问题属于  $NP$ , 但是我们不知道这些问题是否属于  $P$ 。也就是说,我们可以很容易地检查任何给定的解决方案是否有效,但是,尽管付出了相当大的努力,没有人能找到一种可以用于寻找解决方案的有效算法。值得注意的是这些问题都有一个额外的属性,即它们能提供令人信服的证据证明它们都是难以解决的,并且可以证明  $P \neq NP$ 。为了研究这一现象,我们定义如下属性:

**定义:** 对于一个搜索问题  $B$  来说,如果每个搜索问题  $A$  都能多项式地归约成  $B$ , 那么  $B$  是  $NP$  完全的。

这是一个非常强大的命题。也就是说,如果一个问题  $B$  是  $NP$  完全的,那么一旦找到一个能解决这个问题的多项式时间算法就意味着我们能在多项式时间内解决所有属于  $NP$  的问题—— $NP$  会等于  $P$ 。

这个定义让我们能够把对“困难”的定义提升到“难以处理,除非  $P = NP$ ”。如此, $NP$  完全问题是最难的搜索问题。如果任何一个  $NP$  完全问题可以在多项式时间内求解,那么  $NP$  中的所有问题都可以(换言之,  $P = NP$ )。也就是说,研究人员试图为所有的这些问题找到有效的算法,但是他们都失败了,而他们的集体失败可能会被看作证明  $P = NP$  的集体失败。

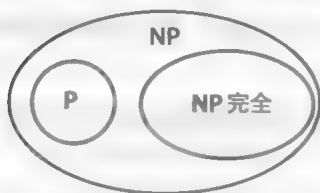
在你看来  $NP$  完全似乎是一个幻想(就像非确定性),但是,正如你将会看到的那样,我们能够证明实际出现的所有搜索问题都属于  $P$  或者  $NP$  完全。大部分的研究者都认为  $P \neq NP$ , 如果能证明一个问题是  $NP$  完全的,那么就意味着这个问题是难以解决的,于是这些研究者就会放弃为这个问题寻找一个多项式时间算法。

843

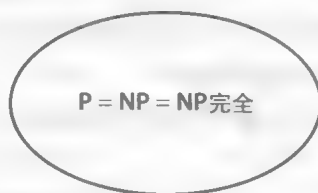
我们仍然生活在两个可能中的一个,但至少我们知道问题的关键在哪。如果一个问题  $B$  是  $NP$  完全的,那么认为它难以处理是合理的。

$P \neq NP$ 

NP 完全问题是难以解决的

 $P = NP$ 

所有搜索问题都是可以解决的



两种可能 (另一角度)

**证明问题是 NP 完全的** 与难解性一样，如果我们不知道哪个问题是 NP 完全的，那么 NP 完全性的概念就没有任何作用。但这正是它们的区别！我们知道很多自然界中的问题是 NP 完全的，而且我们有一个相对直接的方式来分辨新问题是否是 NP 完全的。这个情况与我们看到的可计算性相同（见 5.4 节）。困难的部分是证明第一个这样的问题是 NP 完全的。

**理论**（库克—莱维，1971）：布尔可满足问题是 NP 完全的。

简洁的证明过程如下：我们的目标是证明如果布尔可满足性有一个多项式时间算法，那么所有属于 NP 的问题都可以在多项式时间内解决。给定任何属于 NP 问题的任意实例，证明的第一步都是构造一个不确定性图灵机来解决这个问题，这是由 NP 的定义来决定的。接下来，证明展示了一种用逻辑公式来描述任意图灵机特征的方法，如在布尔可满足性问题中所示。这个描述在每个属于 NP 问题的每个实例（可以表示为一个不确定性图灵机和它的输入）与一些可满足性的实例（把这个机器及其输入翻译成一个逻辑表达式）之间建立了对应关系。最后，可满足性的解决方案与在机器上对给定程序的给定输入进行的模拟是一致的，所以这一个解决方案代表着给定问题的给定实例的解决方案。

这个证明是由史蒂芬·库克和莱昂纳德·莱维在 20 世纪 70 年代初期分别完成的。库克的论文在向人们介绍 NP 完全性的概念方面具有极大的影响力，但是莱维的结果出现得较早，所以习惯上我们把二者结合起来讨论。这个证明进一步的细节超出了本书的范围，但是你可以在以后的计算机科学课程中学到。在目前的学习中，我们只需要知道有这样一个证明就足够了。

就像停机问题的不可解决性一样，布尔可满足性问题虽然非常有趣，但也是不能够以多项式时间解决的。布尔可满足问题的 NP 完全性实在太重要了，因为我们可以用它来证明其他重要问题是不可解决的。

**命题 D**：如果一个问题满足下列条件，那么它是 NP 完全的：

- 这个问题属于 NP（它必须是一个搜索问题）。
- 某个 NP 完全问题 A 可以在多项式时间内归约为这个问题。

**证明**：通过归约的传递性可立即证出。任何属于 NP 的问题都可以在多项式时间内归约为 A，所以它们可以在多项式时间内归约为给定问题。

我们首先将问题 A 看成布尔可满足性问题。如果我们证明布尔可满足性问题可以在多项式时间内归约为问题 B，那么我们就已经证明 NP 中的任何问题都能归约为 B。换句话

说，问题 B 是 NP 完全的。

卡尔普归约。1972 年，理查德·卡尔普 (Richard Karp) 用这种方式展示了由布尔可满足性到 21 个众所周知难以解决的问题的归约过程。作为一个例子，我们再次考虑 0/1 ILP。

**命题 E:** 0/1 整数线性不等式可满足性是 NP 完全的。

证明：我们已经知道 0/1 ILP 是属于 NP 的。所以我们证明布尔可满足性可在多项式时间内归约为 0/1 ILP 就足够了。给定一个布尔可满足性的实例，定义一组不等式，其中每一个 0/1 变量对应一个布尔变量。然后通过将 “= true” 替换为 “ $\geq 1$ ”，用  $(1-x)$  替换每个布尔变量  $x$  的否定形式，从而将每个布尔方程转换为不等式。如果一个 0/1 变量的值为 1，那么含有这个变量的非否定形式的不等式都可以被满足；如果一个 0/1 变量的值为 0，那么含有这个变量的否定形式的不等式都可以被满足。因此，任何 0/1 ILP 实例的解决方案可立即转化为对应布尔可满足性实例的解决方案。

845

布尔可满足性问题

找到满足式子的  
 $x, y, z$  真/假值

$$(x' + z)(x + y' + z)(x + y)(x' + y')$$

布尔可满足性的方程式表达

找到满足式子的  
 $x, y, z$  真/假值

$$\begin{aligned} x + z &= \text{true} \\ x + y' + z &= \text{true} \\ x + y &= \text{true} \\ x' + y' &= \text{true} \end{aligned}$$

0/1 ILP 的表达

找到满足式子的  
 $x, y, z$  的 0/1 值

$$\begin{aligned} (1-x) + z &\geq 1 \\ x + (1-y) + z &\geq 1 \\ x + y &\geq 1 \\ (1-x) + (1-y) &\geq 1 \end{aligned}$$

0/1 ILP 的解决方案

$$\begin{aligned} x &= 0 & x &= 1 \\ y &= 1 & y &= 0 \\ z &= 1 & z &= 1 \end{aligned}$$

布尔可满足性的解决方案

$$\begin{aligned} x &= \text{false} & x &= \text{true} \\ y &= \text{true} & y &= \text{false} \\ z &= \text{true} & z &= \text{true} \end{aligned}$$

将布尔可满足性归约为 0/1 ILP

左图给出了这种构造方式的一个例子。请注意，我们将顶点覆盖问题归约为 0/1 ILP 问题 (命题 C) 并不会产生顶点覆盖问题是 NP 完全的这样的结论，因为我们还没有证明顶点覆盖问题是 NP 完全的。但是我们可以用 0/1 ILP 问题的 NP 完全性来证明 ILP 问题是 NP 完全的。

**命题 F:** ILP 问题是 NP 完全的。

证明：我们已经注意到 ILP 是属于 NP 的。接下来，我们会证明 0/1 ILP 可以在多项式时间内归约为 ILP。给定任何一个 0/1 ILP 的实例，通过对每个变量  $x$  增加  $x \leq 1$  和  $x \geq 0$  这样的不等式来将它转换为一个 ILP 的实例，其中所有变量的值只能为 0 或者 1。这样任何 ILP 实例的解决方案都会立即变为 0/1 ILP 问题的一个解决方案。

综上所述：一个解决 ILP 问题的多项式时间算法可以给出 0/1 ILP 问题的多项式时间算法，而 0/1 ILP 问题的多项式时间算法又会给出布尔可满足性问题的多项式时间算法，布尔可满足性问题的多项式时间算法可以给出所有属于 NP 问题的多项式时间算法 ( $P = NP$ )。在卡尔普的论文和库克-莱维定理出现之前，人们知道 ILP (作为例子) 是困难的，但是他们不知道由一个多项式时间算法给出所有搜索问题的多项式时间算法如此困难。

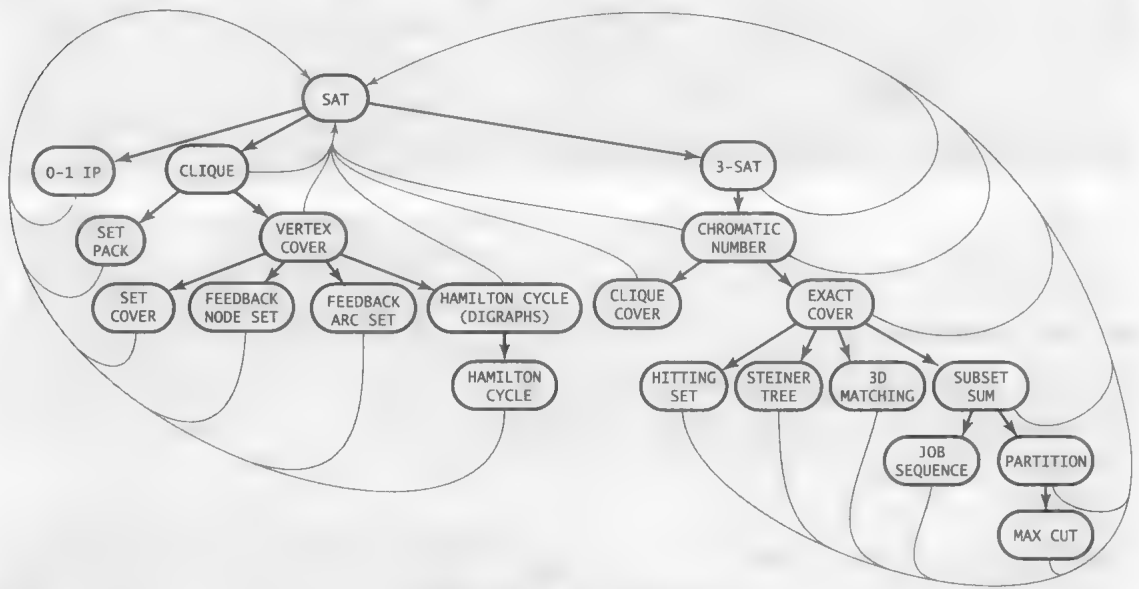
现在你已经知道三个 NP 完全问题，并且可以通过归约它们中的任何一个来找到另一个 NP 完全问题。你可能会看到这些已知的 NP 完全问题如何迅速扩张。

卡尔普 1972 年的论文《Reducibility Among Combinatorial Problems》中提到的这个问题我们将在后文描述，他对归约的证明会在下图展示，其中从问题 A 到问题 B 的箭头表示

846



在论文中已经证明了可以从 A 到 B 进行多项式时间的归约。请注意，顶点覆盖问题也在其中，从 SAT 到 CLIQUE 的归约和从 CLIQUE 到 VERTEX COVER（顶点覆盖）的归约都被证明是 NP 完全的。图中的蓝色箭头阐述了库克—莱维定理的含义，它向我们强调所有问题都是 NP 完全的。这篇论文有很大的影响力，因为这是一个明确的陈述，如果所有这些问题都有多项式时间算法，那么  $P = NP$ 。人们对于这些问题的困难之处各有看法，但是事实上它们都非常困难，并且其中的任何一个问题的有效算法都意味着是对所有问题的有效算法的一个启示。从此人们意识到为这些问题寻找多项式时间算法是没有意义的，有许多研究人员已经花了很多年时间去试图解决这些问题。



库克 - 莱维 - 卡尔普归约

坍塌式发展。自卡尔普发表论文后的几十年来，研究人员已经通过这种归约关系在各种各样的领域发现了数以万计的问题。在下面的表格中列出了其中一些例子。显然，与不可解决性相反，已知的 NP 完全问题的数量正在迅速增加。每年都有数千篇关于这个话题的科学论文被发表出来。正因为有这么多学科在这方面有如此多的科学问题， $P = NP$ ？毫无疑问是我们这个时代最重要的开放性科学问题之一。

847

| 问题                    | 简洁描述                                        |
|-----------------------|---------------------------------------------|
| 布尔可满足性 ...<br>( SAT ) | 同时求解一些布尔方程                                  |
| 3-SAT                 | 同时求解一些布尔方程，一条方程最多有三个变量                      |
| 0/1 ILP               | 同时求解线性不等式，并且取值限制在 0/1                       |
| 团                     | 找到一个至少有 $m$ 个顶点的完全子图                        |
| 集合的包                  | 从给定列表中找出 $m$ 个或者更少的成对不相交的子集                 |
| 顶点覆盖                  | 寻找 $m$ 个或者更少的顶点使得这些顶点能接触图中的所有边              |
| 集合覆盖                  | 在一个给定的列表中寻找 $m$ 个或者更少的 $S$ 的子集，使得它们的并集是 $S$ |
| 反馈顶点集合                | 在一个图中寻找至多 $m$ 个顶点，将这些顶点全部移除后图中不存在环          |

卡尔普的 NP 完全问题的简洁描述（表达为搜索问题）

|         |                                                            |
|---------|------------------------------------------------------------|
| 反馈边集合   | 在一个图中寻找至多 $m$ 条边, 将这些边全部移除后图中不存在环                          |
| 施泰纳树    | 用总长不超过 $m$ 的直线连接一个集合中的所有点, 并且允许不在原始集中额外的“施泰纳点”存在           |
| 有向哈密顿回路 | 在一个有向图中找到一个有向循环, 使得每个顶点都被经过一次                              |
| 哈密顿回路   | 在一个图中找到一个回路, 使每个顶点都被经过一次                                   |
| 图着色     | 使用 $m$ 种或者更少的颜色为图中的每个顶点着色, 并且每条边连接的两个顶点颜色不同                |
| 团覆盖     | 将一个图分成 $m$ 个或更少的团, 或者报告不可能                                 |
| 精确覆盖    | 从给定列表中找出至多 $m$ 个 $S$ 的子集, 使得每个 $S$ 的成员在且仅在一个子集中, 或者报告不可能   |
| 命中集合    | 寻找一个包含至多 $m$ 个元素的 $S$ 的子集, 其中包含给定列表中的每个子集的至少一个元素           |
| 三维匹配    | 给定一个三元组的集合, 其中每个三元组的元素都来自三个不相交的集合, 找到一个子集使得没有一个元素出现在两个三元组中 |
| 背包      | 与我们对子集求和问题的描述相同                                            |
| 作业车间调度  | 在两个处理器上对一组作业进行调度, 使得在指定的时间 $m$ 内将所有作业完成                    |
| 分块      | 将一个整数集合分成两个和相等的子集                                          |
| 最大分割    | 找到一个最多有 $m$ 条边的集合将图分割为不相交的两块                               |

卡尔普的 NP 完全问题的简洁描述 (表达为搜索问题)(续)

848

| 研究领域  | 典型 NP 完全问题          |
|-------|---------------------|
| 航天工程  | 有限元素的最优网格划分         |
| 生物    | 发展重构                |
| 化学工程  | 热交换网络合成             |
| 化学    | 蛋白质折叠               |
| 土木工程  | 城市交通流量均衡            |
| 计算机设计 | VLSI 布局             |
| 经济学   | 金融市场摩擦的仲裁           |
| 环境科学  | 污染物传感器的最佳位置         |
| 金融工程  | 最大限度减小投资风险          |
| 博弈论   | 使社会福利最大化的纳什均衡       |
| 基因组学  | 基因重构                |
| 机械工程  | 剪切流中的湍流结构           |
| 药学    | 双翼心血管的图像重构          |
| 行为调查  | 旅游销售人员、整数编程等        |
| 物理学   | 三维易辛模型的分区功能         |
| 政治学   | Shapley-Shubik 投票理论 |
| 热门文化  | 数独、跳棋、扫雷、俄罗斯方块      |
| 统计学   | 最佳的实验设计             |

NP 完全问题的例子

849

尽管 NP 完全理论没有提供任何可以证明这些问题是难处理的证据, 但是它还是有着很深远的影响; 它将任何一个对于这些问题的多项式时间算法发展成为与证明  $P = NP$

等价的问题。它们是同一个问题的不同表现形式！寻找一个有效的（多项式时间）算法来进行基因重构，或者设计一个计算机芯片来使得投资组合的风险最小，就等于是要证明  $P = NP$ 。

**应对 NP 完全性** 与可计算性一样，NP 完全是当今世界不可避免的问题。正如我们在讨论可计算性时提出的那样，如同我们在 5.4 节和本节中看到的很多例子，只有自己亲身参与过计算的人才会倾向于认为我们可以用一台足够强大的计算机做任何事，这个假设毫无疑问是错误的。那些不了解 NP 完全的人注定会遇到令人沮丧的体验，即他们试图为一个已经被证明（或者容易被证明）为 NP 完全的问题开发一个算法，但是他们却不知道这样的算法将会是一个著名的开放性问题的解决方案。如果你需要应对 NP 完全的问题，那么你首先要理解一个给定的问题是如何被分类的，然后在实践中采取适当的策略来解决这个问题。



经 Nokia 公司许可转载

850

### 难解性理论的一个实际用处

问题的分类。在实践中，我们的优势在于面临一个新的问题时，我们可以在两个选择中选择一个：

- 证明这个问题属于 P。
- 证明这个问题是 NP 完全的。

虽然还有其他的选择，但是除了这个问题不是搜索问题或者这个问题是无法解决的，你在实践过程中是不可能遇到其他选择的（请参考本节问答环节）。

为了证明一个问题属于 P，我们需要找到一个多项式时间算法来解决这个问题，也许可以通过多项式时间将其归约为一个已知属于 P 的问题。这个过程与编写使用现有库的 Java 程序没有区别——如果已知问题可以在多项式时间内执行，那么客户程序也可以在多项式时间内执行。一旦我们知道一个问题属于 P，我们就可以开发改进的算法，正如我们在书中对许多问题所做的那样。

为了证明一个问题属于 NP 完全的，我们需要证明这个问题属于 NP，并且某个已知的 NP 完全问题可以在多项式时间内归约成它。也就是说，新问题的一个多项式时间算法可以用来解决 NP 完全问题，而这个 NP 完全问题的解决方案又可以用来解决所有属于 NP 的问题。我们已知的成千上万个被证明是 NP 完全的问题都是从一个已知是 NP 完全的问题归约得到的，就像我们在命题 E 中对 0/1 ILP 和命题 F 中对 ILP 所做的那样。

从实际的角度来看，将问题分类为易于解决 (P) 和难以解决 (NP 完全) 的过程如下：

- 直截了当的。例如，插入排序证明排序算法是属于 P 的。
- 棘手但是不困难。例如，证明 0/1 ILP 是 NP 完全的（命题 E）需要一些经验和实践，

但是很容易理解。

- 极具挑战性。例如，线性规划在 20 世纪 80 年代被证明属于  $P$  之前长期处于未被分类的状态。
- 开放性。例如，图同构（给定两个图，找到一个方法来对其中一个图的顶点进行重命名，使得两个图相同）和因式分解问题（给定一个整数，找到一个非平凡的因数分解式）仍然是未被分类的。

这是目前研究的一个丰富且活跃的领域，每年都有数千篇研究论文。正如在前面表格中指出的那样，科学探究的所有领域都与之有关联。回想一下，我们对  $NP$  的定义涵盖了科学家、工程师和应用程序员都希望能够切实解决的问题——当然所有这些问题都需要分类！

851

解决  $NP$  完全问题的策略。面对如此广泛的问题，人们必须要取得一些进展，所以人们投入了很多精力以想办法解决这些问题。难解性理论告诉我们，世界上存在着很多重要问题，我们不能合理地期望找到一个算法以同时满足下面三个属性：

- 保证用最优的方式解决这个问题（针对优化问题）。
- 保证在多项式时间内解决问题。
- 保证解决这个问题的任何实例。

因此，当我们遇到  $NP$  完全问题时，我们必须至少放弃这三个要求中的一个。我们不可能在短短的几个段落之内对如此广阔的领域做出准确而公正的评论，但是我们简要地描述了两种成功的方法。

近似 (Approximate): 放宽最优性要求等于改变问题并开发一种近似算法 (approximation algorithm)，这个算法不一定是最好的解决方案，但是我们能保证这种算法接近最好的解决方案。例如，我们可以很容易地找到顶点覆盖问题的优化版本的解决方案，它的代价在最优方案的两倍的范围内（见练习 5.5.41）。设计近似算法是一个活跃的研究领域。不幸的是，当我们对近似算法做进一步的改进时会发现，这种方法通常是无效的，它仍然没有办法回避  $NP$  完全问题。例如，我们不可能找到这样的一种情况，对于一个特定问题，如果你能找到一个近似算法来保证它的结果能够确保在最优性能两倍范围内，那么  $P = NP$ 。设计这种  $NP$  完全问题的近似算法是不可能的，除非  $P = NP$ 。

忽略最坏情况后的性能保证 (Ignore the worst-case performance guarantees): 放宽其他两个要求之一就是开发一种算法，这种算法可以有效地解决在实际中出现的典型情况，即使这种算法存在最坏输入情况下找不到解决方案的可能。在这种情况下，我们不再保证问题的任何输入都能在多项式时间内被解决，而是保证对于实际到达这个问题的输入都可以用多项式时间来解决。同时，我们也不再保证这个算法可以解决问题的任意实例，而是让程序知道这个问题的哪些实例需要用指数级时间来解决。在现代工业应用中经常使用 SAT、ILP 和其他  $NP$  完全问题的解决器来解决出现的巨大问题。我们会对让一个求解器运行缓慢的具体问题进行深入研究，直到找到一些成功的方法来处理它们。接下来我们将讨论 SAT 的这种方法。

852

布尔可满足性。为了给本节做一个总结，我们将开发一个算法来解决著名的 SAT 问题的实例。我们这样做可以让你对计算机科学领域的研究人员几十年来一直在争论的一个非常重要的问题有所了解。我们是否应该将 SAT 看作一个易于解决的问题？

开始之前，我们需要有一些用于表示问题实例和潜在解决方案的约定。我们用  $m$  表示

方程式的数量，用  $n$  表示变量的数量。为了表示一个解决方案，我们使用了一个长度为  $m$  的布尔型数组 `inSubset[]`，它表示了赋值为 `true` 的变量的子集。为了表示一个问题实例，我们使用右侧展示的紧凑编码。我们将每个方程表示为含有  $n$  个字符的字符串，其中字符串中的第  $i$  个字符对应于第  $i$  个变量，如果变量出现在（非否定）方程中，则为“+”，如果变量出现在（否定）方程中，则为“-”，如果变量没有出现在方程中，则为“.”。因此，每个问题实例都可以用一个包含  $m$  个字符串的数组 `clauses[]` 来表示。

通过这种表示形式，我们很容易就能检查一个给定的解决方案 `inSubset[]` 是否满足所有方程：

```
public static boolean check(String[] clauses, boolean[] inSubset)
{
 boolean product = true;
 for (int i = 0; i < clauses.length; i++)
 {
 boolean sum = false;
 for (int j = 0; j < inSubset.length; j++)
 {
 if (clauses[i].charAt(j) == '+') sum = sum || inSubset[j];
 if (clauses[i].charAt(j) == '-') sum = sum || !inSubset[j];
 }
 product = product && sum;
 }
 return product;
}
```

问题：

```
x' + z = true
x + y' + z = true
x + y = true
x' + y' = true
```

输入格式：

```
-.+
+--+
+++.
---.
```

SAT 表示

代码中的 `check()` 函数可以在与问题大小成线性的时间内完成，因此我们可以确定 SAT 属于 NP。它也是我们接下来描述的 SAT 求解器中的关键子程序。

为了找到一个 SAT 实例的解决方案，我们可以枚举所有  $2^n$  个可能的赋值，然后使用 `check()` 函数来识别哪种赋值方式满足所有等式。SAT（程序 5.5.1）通过对 `inSubset[]` 数组中所有可能的赋值进行“计数”来完成这个操作，其中 `false` 对应 0，`true` 对应 1，整个数组对应一个二进制数（见“子集和问题的一个示例”图表）。为了强调即使是对于图灵机这也是一个简单的计算，SAT 中 `next()` 方法使用的算法与我们在增量图灵机中使用的算法相同，采用这种算法对 `inSubset[]` 进行“递增”来得到下一个值的集合：从右侧开始扫描，将 `false` 的值改为 `true`，直到遇到一个 `true` 并将它改成 `false`。如果没有找到 `true`，算法终止，在这种情况下 `next()` 函数返回 `false`；否则它返回 `true`。构造方法通过调用 `next()` 函数来精心推断计算下一个解决方案，通过调用 `check()` 来检查是否确实是一个解决方案，持续运行直到 `next()` 返回 `false`。`main()` 函数从指定为命令行参数的文件中读取数据，并创建一个 SAT 对象来解决可以满足的实例。

```
public static void main(String[] args)
{
 String filename = args[0];
 In in = new In(filename);
 String[] clauses = in.readAllStrings();
 SAT solver = new SAT(clauses);
 StdOut.println(solver);
}
```

这是一个经典而且基本的编程练习，还有很多其他的可行方法，这些方法都与我们在本书其他地方介绍过的话题相关。这些实现是相当有趣的，它们之中没有一个与前文讲到的计算理论存在差异，所以我们只在练习中描述其中的一部分（见练习 5.5.15、练习 5.5.16 和练习 5.5.17）。



都需要指数时间来求解。例如，程序 5.5.1 会对一个不存在解决方案的实例检查所有的  $2^n$  种可能的分配方案。出现在真实世界的应用程序中的实例是否更适合随机模型，而非最坏情况模型呢？没有人知道。在计算理论的背景下，我们除了将 SAT 求解器分类为指数时间的算法外别无选择。

856

这种情况十分令人着迷，所以在这里需要强调我们在本节讨论的理论都是基于最坏情况的分析。难解性理论的主要思想为我们对计算的理解做出了很大的贡献，但是现实世界的程序员已经找到了避免最坏情况性能的方法。

本书研究的所有应用领域都与 NP 完全问题存在密切联系。在基础编程、排序和搜索、图形处理、字符串处理、科学计算、系统编程以及任何可以想象到涉及计算的领域都存在 NP 完全问题。很少有科学理论能有如此广泛而深远的影响力。

对于上一节描述的可计算性问题来说，它们有着严谨的证明过程，因此很少有人会去怀疑邱奇-图灵假设，而且我们也确实已经证明有一些问题是不可解的。对于本节讨论的难解性问题，仍然有两个地方让我们无法完全相信它们就是事实。首先，量子计算已经引起一些人怀疑扩展的邱奇-图灵理论（但是仍然没有证据表明，如果我们能够从物理上制造一台量子计算机计算设备，我们就可以在多项式时间内解决 NP 完全问题）。其次，没有人能证明一个单一的搜索问题是难以解决的——这将证明  $P \neq NP$ 。

即使如此，NP 完全问题的存在极大地扩展了所有计算机都存在内在限制的观点。这些问题具有重大的现实意义，但是鉴于目前的知识状况，我们必须认识到我们无法保证能有效地解决这些问题。

857

## 问答环节

问：多项式时间算法总是有效的吗？

答：不，需要使用  $n^{100}$  或者  $10^{100}n^2$  步的算法在实际应用中和指数级时间算法一样是无用的。但是实际中出现的常量通常是足够小的，所以用  $P$  来代表“在实践中有用的算法”是合理的。

问：指数级时间算法总是无用的吗？

答：不。例如当  $n$  小于一百万的情况下，运行时间为  $1.000\ 01^n$  的算法很可能是十分有用的。同样的，SAT（程序 5.5.1）对于许多常规输入来说是十分有效的，其中包括我们分析过的随机生成的输入。

问：为什么对多项式时间算法的定义（用最坏运行时间的上限定义）与指数级时间算法的定义（用最坏运行时间的下限定义）有所不同？

答：通常情况下，我们试图将“易于解决”的问题和“难以解决”的问题相区别。对于易于解决的问题，我们寻求多项式时间的上限，而对于难以解决的问题我们寻求指数时间的下限。所以，当我们提出一个指数级时间算法时，我们的意思是（对于一类无限多的输入）它运行所需要的时间最少是指数级时间，而非最多。

问：为什么一个指数级时间算法的定义要求算法对于无限多的输入需要花费指数时间，而不是对于所有输入。

答：这是一个合理的替代定义（一些计算机科学家使用更严格的版本）。我们没有采用更严格的版本，是因为我们认为在无限多的输入的情况下，任何需要指数时间的算法都是低效的。

858



问：是否存在这么一个问题，可以证明这个问题的最好算法需要指数时间，但是这个问题独立于  $P = NP$  问题之外。

答：是的。确实存在极少数的自然问题需要指数时间。以下版本的停机问题是一个值得注意的例子：给定一个没有输入的图灵机（或者 Java 程序）和整数  $k$ ，这个图灵机（或者这个 Java 程序）会在少于  $k$  步的时候停止吗？你只需要在这个图灵机（或者这个 Java 程序）上运行最多  $k$  步，直到它停止或者完成  $k$  步。研究人员已经证明没有比穷举法模拟更好的办法。这种穷举法模拟花的时间是输入大小的指数量级，因为输入  $k$  可以用  $\lg k$  位进行二进制编码。

问：有一些问题的算法既不是多项式时间函数也不是指数时间函数，如  $n^{\log n}$ ，我们要怎么对这些问题进行分类呢？

答：好问题。这个问题的一个著名的例子是图同构（graph isomorphism）问题，即给定两个图，判断这两个图除了顶点的名称以外是否等价。这个问题属于  $NP$  但并不被认为（或被相信）是  $NP$  完全的。科学家在 2015 年才找到一个能以  $n^{\log n}$  时间运行的算法。如果我们在某天能找到一个多项式时间算法来解决图同构问题，那也并不能表示  $P = NP$ 。

问：属于  $NP$  的问题中有既不属于  $P$  也不是  $NP$  完全的吗？

答：是的。在  $P \neq NP$  的假设下，理查德·拉德纳（R. Ladner）在 1975 年证明了  $NP$  中存在既不属于  $P$  也不是  $NP$  完全的问题。一些研究人员怀疑分解问题和图同构问题即属于此类复杂的问题。

问：我听说过一个复杂的等级叫“ $NP$  难”（NP-hard）问题，那是什么？

答：如果所有属于  $NP$  的问题都能归约成某个问题，那么这个问题是  $NP$  难的。这个定义与“ $NP$  完全”是相同的，但是不要求这个问题属于  $NP$ 。

859

问：为什么现代密码系统不是基于  $NP$  完全或  $NP$  难问题而是因式分解？

答：研究人员试图做到这一点。但是  $NP$  完全考虑的是复杂性的最坏情况，而密码系统应用需要在实际中对于所有的输入都是难以解决的。

问：我可以从哪里学到更多有关  $NP$  完全的知识？

答：有一本经典的参考文献是 Garey 和 Johnson 写的《计算机与难解性：NP 完全性理论指南》。这本书最近被列为计算机科学文献中被引用次数最多的参考文献。许多重要的后续发现都被记载在《算法杂志》（Journal of Algorithms）中的 Johnson 的 NP-completeness 栏目里。

860

## 练习

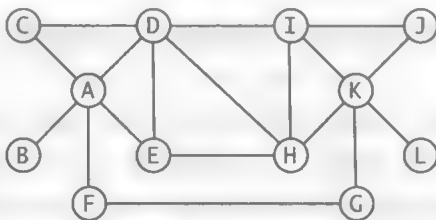
5.5.1 填写下表中的空格（对于较大的数字以 10 为底数）。

| $n$    | $(1.1)^n$ | $n^4$  | $2^n$     | $n!$       |
|--------|-----------|--------|-----------|------------|
| 10     | 2.59...   |        | 1,024     | 3,628,800  |
| 100    | 13,781    | $10^8$ | $10^{30}$ | $10^{158}$ |
| 1,000  | $10^{41}$ |        |           |            |
| 10,000 |           |        |           |            |

5.5.2 编写一个可以实现高斯消元法的 Java 程序。用户需要输入  $n$  行信息（每行对应一个方程），每行

都有  $n + 1$  个双精度值（变量的系数和等式右侧的值）。首先假设方程组存在唯一的解，并根据文中给出的例子调试程序。然后讨论可能出现的问题和应对的策略。

- 5.5.3 如果有一些算法对于无限输入花费的时间与  $1.5^n$  成比例，那么将这些算法描述为指数级时间算法是否合适？解释你的答案。并将花费的时间替换为  $n!$ 、 $n^4$ 、 $2\sqrt{n}$  和  $n^{\log n}$ ，再次回答问题。
- 5.5.4 编写一个程序，这个程序可以从标准输入中读取  $n$  个整数的序列，并找到总和为 0 的非空子集（或者报告不存在这样的子集）。你需要穷举  $n$  个整数的所有  $2^n$  个子集。
- 5.5.5 为下图找到一个最少元素的顶点覆盖解决方案：



861

- 5.5.6 当：(1)  $m$  是常数  $k$ ；(2)  $m$  大约为  $n/2$  时，对于一个具有  $n$  个顶点的图，估略计算至多有  $m$  个顶点的子集数量。
- 5.5.7 编写一个方法，这个方法以图  $G$  和一个顶点的子集为输入，并确定这个顶点的子集是否为一个顶点覆盖。
- 5.5.8 证明任何布尔可满足性问题都可以转化为一种形式，这种形式的左侧不使用与操作符且右侧都为真。提示：参见 7.1 节中布尔函数中关于求和的定义。
- 5.5.9 为 SAT 实现一个 `toString()` 方法（程序 5.5.1）。
- 5.5.10 为 SAT 问题开发一个使用二维整数数组表示的方法。将正文中的数组 `string` 中的 `+1`、`-1` 和 `0`，用 `“+”`、`“-”` 和 `“.”` 来替换。通过为你的表示开发一个多项式时间的 `check()` 方法来证明 SAT 属于 NP。
- 5.5.11 证明 0/1 ILP 属于 NP。
- 5.5.12 考虑以下经典问题：

旅行商问题（Traveling salesperson, TSP）：给定一组  $n$  个城市的集合和一个距离  $m$ ，找到一个长度小于或等于  $m$  的遍历所有城市的旅行方法，或者报告不存在这种旅行方法。

为了避免比较整数平方根之和的技术问题，我们假定所有距离都是整数（采用欧几里得距离表示，单位为英里或米，取相近的整数）。通过开发一个多项式时间的 `check()` 方法来证明 TSP 属于 NP，这个方法可以检查一个给定旅行方法是否能够满足给定距离的限制。假设 `check()` 的参数是一个整数  $n$ （城市的数量）、一个整数  $m$ （描述旅行方法的长度上限）、两个整数数组 `x[]` 和 `y[]`（点的  $x$ 、 $y$  轴坐标），以及一个整数数组 `tour[]` 用于指定城市出现在旅行方法中的顺序。

862

- 5.5.13 编写一个名为 `GenerateSAT` 的程序，这个程序可以生成随机的 SAT 实例，格式如文中所述。程序需要以下四个命令行参数：

- $m$ ，方程的数量
- $n$ ，变量的数量
- $p$ ，非负变量的百分比
- $q$  负变量的百分比

答案:

```
public class GenerateSAT
{
 public static void main(String[] args)
 {
 int m = Integer.parseInt(args[1]);
 int n = Integer.parseInt(args[0]);
 double p = Double.parseDouble(args[2]);
 double q = Double.parseDouble(args[3]);
 for (int k = 0; k < m; k++)
 {
 String equation = "";
 for (int i = 0; i < n; i++)
 {
 double x = StdRandom.uniform();
 if (x < p) equation += "+";
 else if (x < p+q) equation += "-";
 else equation += ".";
 }
 StdOut.println(equation);
 }
 }
}
```

正文中使用的 SAT30-by-30.txt 文件就是由该程序创建的——使用命令 “java GenerateSAT 30 30 0.1 0.1” 创建。

5.5.14 以各种各样的参数运行练习 5.5.13 中的 GenerateSAT 程序，试图找到一个 SAT 实例，这个 SAT 实例有 30 个变量，这些变量可以使得 SAT 程序中执行测试的次数尽可能增多。 863

5.5.15 在 6.1 节中，我们描述了 Java 对 int 类型值的二进制表示的操作，并引入了如下代码：

- “ $1 \ll n$ ” 可以计算  $2^n$ 。
- “ $(v \gg i) \& 1$ ” 是  $v$  的二进制表示从右侧数起的第  $i$  位。

基于这些代码片段实现一个 SAT。

5.5.16 基于一个像 TowersOfHanoi (程序 2.3.2) 的递归函数实现一个 SAT：将 inSubset[] 中最右边的值设置为 false，然后调用递归函数来生成所有剩余可能的值，然后将最右边的值设置为 true 并进行递归调用，以便可以再次生成所有剩余可能的值。对于得到的每一种情况再调用 check()。

5.5.17 基于一个像 Beckett (程序 2.3.3) 的递归函数实现一个 SAT。对于某些应用，这种方法比练习 5.5.16 中的方法更受欢迎，因为子集的大小每次最多只会改变 1。

5.5.18 使用 SAT 中的代码和前文 5.5 节中给出的 check() 方法作为基础，开发一个 SubsetSum 程序，这个程序可以为任何给定的数字集合生成子集并寻找问题的解。

5.5.19 使用 SAT 中的代码和练习 5.5.7 中的 check() 方法作为基础，开发一个 VertexCover 程序，这个程序可以为任何给定的图找到顶点覆盖问题的解（上限为覆盖的顶点数）。

5.5.20 将 SAT 中子集生成部分的代码封装到自己的类中，然后基于这个类实现 SAT (程序 5.5.1)、SubsetSum (练习 5.5.18) 和 VertexCover (练习 5.5.19)。

5.5.21 描述一些需要 SAT 计算  $n$  个变量的所有  $2^n$  种情况的实例。

答案：使用  $n$  个方程，其中第  $i$  个方程中仅包含第  $i$  个变量（非负）。只有当所有的变量都为 true 时，这些方程才能同时被满足，这是最后一个被 SAT 检测的条件。 864

5.5.22 假设有两个问题已知是 NP 完全的。这是否意味着这两个问题中的一个可以通过多项式时间归

约为另一个。

5.5.23 假设  $X$  是 NP 完全的,  $X$  可以归约成  $Y$ ,  $Y$  也可以归约成  $X$ 。Y 必然是 NP 完全的吗?

答案: 不,  $Y$  可以不属于 NP。

5.5.24 如果  $P \neq NP$ , 那么是否有一个算法能在  $n \log n$  的时间范围内解决 NP 完全问题? 解释你的答案。

5.5.25 假设有人发现了一个保证能在正比于  $1.1^n$  的时间内解决布尔可满足性问题的算法。这是否意味着我们可以在正比于  $1.1^n$  的时间内解决其他 NP 完全问题。

5.5.26 一个能在正比于  $1.1^n$  的时间内解决顶点覆盖问题的程序有什么重大意义?

5.5.27 我们假设  $P \neq NP$ , 那么下面选项中的哪一个可以从 TSP 是 NP 完全的事实中推断出来?

- a. 不存在能解决任意 TSP 实例的算法。
- b. 不存在能有效解决任意 TSP 实例的算法。
- c. 存在一种能有效解决任意 TSP 实例的算法, 但是没有人能找到它。
- d. TSP 不属于 P。
- e. 对于某些输入, 所有算法都能保证在多项式时间内解决 TSP。
- f. 对于所有输入, 所有算法都能保证在指数级时间内解决 TSP。

答案: 只有 b 和 d。

5.5.28 我们假设  $P \neq NP$ , 且: 因式分解属于 NP, 但我们并不知道它是 P 还是 NP 完全的, 那么, 可以推断出以下哪个观点?

- a. 存在一个可以解决因式分解任意实例的算法。
- b. 存在一种算法可以有效地解决因式分解的任意实例, 但是没人能找到它。
- c. 如果我们找到一个有效的因式分解算法, 我们可以用它来解决 TSP 问题。

5.5.29 解释为什么以下问题都不是 NP 完全的。

- a. TSP 的穷举法。
- b. 归并排序。
- c. 停机问题。
- d. 希尔伯特的第十个问题。

答案: NP 完全描述的是问题而不是问题的具体算法, 所以将选项 a 和 b 描述为 NP 完全是不正确的 (苹果和桔子也不是 NP 完全的)。停机问题和希尔伯特的第十个问题是不可判定的, 所以它们不属于 NP, 因此也不是 NP 完全的。

5.5.30 证明多项式时间的归约是可以传递的。即证明如果 A 可以在多项式时间内归约成 B, B 可以在多项式时间内归约成 C, 则 A 可以在多项式时间内归约成 C。

5.5.31 A 和 B 是两个决策问题。假设我们知道 A 可以在多项式时间内归约成 B。我们可以推导出以下结论的哪一个?

- a. 如果 B 是 NP 完全的, 那么 A 也是。
- b. 如果 A 是 NP 完全的, 那么 B 也是。
- c. 如果 B 是 NP 完全的且 A 属于 NP, 那么 A 是 NP 完全的。
- d. 如果 A 是 NP 完全的且 B 属于 NP, 那么 B 是 NP 完全的。
- e. A 和 B 不能同时是 NP 完全的。
- f. 如果 A 属于 P, 那么 B 属于 P。
- g. 如果 B 属于 P, 那么 A 属于 P。

答案：只有  $d$  和  $g$ 。

866

5.5.32 证明在有向图中找到哈密顿回路的问题是 NP 完全的。可以通过对在无向图找到哈密顿回路的算法进行归约来解决。

5.5.33 假设我们有一个算法可以解决布尔可满足性问题的决策版本——对于任意输入来说，它可以确定是否存在一个变量的真值分配来满足布尔表达式。演示如何用这种算法来找到一个分配方案。

5.5.34 假设我们有一个解决顶点覆盖问题的算法——对于任意的图和任意的整数  $m$ ，它可以判定是否存在最多只有  $m$  个顶点的顶点覆盖或者报告不存在。展示如何使用这样一个算法来解决问题的优化版本——给定任意的图，找到一个顶点个数最少的顶点覆盖方案。

5.5.35 解释为什么顶点覆盖问题的优化版本不是一个搜索问题。

答案：没有办法来验证一个新提出的解决方案是最好的。顶点覆盖的决策版本是一个搜索问题（当距离都是整数时），因为我们可以使用二分搜索来找到最佳的解决方案。

867

## 创新练习

5.5.36 因式分解。通过修改 Factors（程序 1.3.9）来使用 BigInteger，并使用它来因式分解  $1111111111$ ,  $11111111111$ ,  $\dots$ ,  $(10^n-1)/9$ ,  $\dots$ ，尽可能优化你的程序，看它能不能在 10 秒的运行时间内完成因式分解。

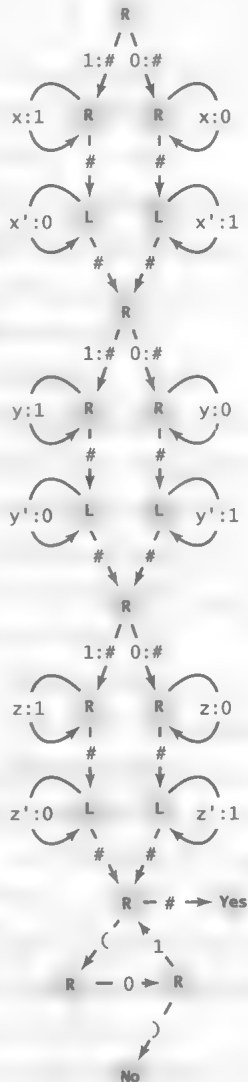
5.5.37 因式分解决策问题。下面显示了如何将因式分解的决策版本转化为一个搜索问题。具体来说，给定一个静态的方法  $\text{factor}(x,y)$ ，如果  $x$  有小于  $y$  的非平凡因子，这个方法返回 true，编写一个静态的方法  $\text{factor}()$  来打印出  $x$  的因子。

5.5.38 SAT 检查器图灵机。开发一个 TM，这个 TM 可以检查一组给定的值是否满足一个给定的 SAT 实例。假设有三个变量，字母表为 “# 0 1 x y z x' y' z' ( ) #”，而且纸带的初始内容是要检查的  $x$ 、 $y$ 、 $z$  的值，后面跟 SAT 实例的简短表示形式。例如，文中给出的例子中的纸带里含有：

# 0 1 1 (x' + z)(x + y' + z)(x + y)(x' + y') #

答案：见右侧画出的 TM。对于每一个变量，它将读取一个值，在向右的扫描中对表达式中的每一个变量进行值的替换，然后在向左的扫描中将表达式中每一个带有否定符号的变量的值进行替换。在完成所有替换之后，TM 底部的五个状态会扫描整个表达式来寻找是否有某一对括号中所有变量的值都不为 1。如果找到了这样一个集合，它进入一个 Yes 状态。如果没有找到，则它进入一个 No 状态。这个解决方案很容易扩展到  $n$  个变量，但并不能证明 SAT 属于 NP。要想完成证明，需要一个与  $n$  无关的更复杂的图灵机结构。

5.5.39 SAT 检查器模拟。制作一个文本文件，这个文件可以为练习 5.5.38 中的通用 DFA TM 提供一个表格表示法。从本书网站上下载 TuringMachine.java 和 Tape.java，按照练习 5.2.7 和练习 5.2.8 中所述进行修改，然后生成这个机器对于指定输入的运行轨迹。



868

答案:

```
0 1 1 (x' + z) (x + y' + z) (x + y) (x' + y') # 从左边开始
1 1 (x' + z) (0 + y' + z) (0 + y) (x' + y') # 将x替换为0
1 1 (1 + z) (0 + y' + z) (0 + y) (1 + y') # 将x替换为1
1 (1 + z) (0 + y' + z) (0 + 1) (1 + y') # 将y替换为1
1 (1 + z) (0 + 0 + z) (0 + 1) (1 + 0) # 将y替换为0
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 将z替换为1
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 将z替换为0
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找“(”或者“#”
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找在“(”之前的1
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找“(”或者“#”
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找在“(”之前的1
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找“(”或者“#”
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找在“(”之前的1
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找“(”或者“#”
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找在“(”之前的1
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 查找“(”或者“#”
(1 + 1) (0 + 0 + 1) (0 + 1) (1 + 0) # 接受
```

- 5.5.40 子集求和 (动态规划解决方案)。开发一个名为 SubsumDP 的 Java 程序, 这个程序可以从标准输入读取整数, 然后尝试所有的可能性来找到一个数字之和为 0 的子集。使用下列方法: 使用一个二维布尔数组 subset[i][j], 若指定: 如果前 j 个数的和为 i, 则 subset[i][j] 为 true, 你的程序的运行时间为多长? 解释你的程序为什么不能证明  $P = NP$ 。

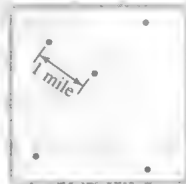
- 5.5.41 顶点覆盖问题的近似算法。考虑顶点覆盖的以下算法: 移除一条边, 并将这条边的终点加入覆盖的集合中, 然后移除在这个终点上的所有边, 不断迭代直到没有剩下的边。证明这样产生的顶点覆盖的结果规模不可能超过最优解的 2 倍。

- 5.5.42 佩尔方程。编写一个 BigInteger 的客户程序 Pell, 这个客户程序可以读取一个整数  $c$ , 并且可以找到佩尔方程:  $x^2 - cy^2 = 1$  的最小解。对于  $c = 61$ , 最小的解为 (1, 766, 319, 049, 226, 153, 980)。对于  $c = 313$ , 最小的解为 (3, 218, 812, 082, 913, 484, 91, 819, 380, 158, 564, 160)。这个问题已经证明无法在多项式 (作为输入  $c$  的位数的函数) 步数内求解, 因为输出可能需要指数级位数!

- 5.5.43 欧几里得 TSP。开发一个 Java 程序 TSP, 这个程序可以通过尝试所有的可能性来找到一个最小长度的 TSP 旅行方法 (见练习 5.5.12)。使用一个递归程序来维护一个标志当前路径上所有城市的数组, 并尝试所有可能的下一个城市。注意: 对于很大的  $n$ , 你的程序将不会结束, 因为  $(n-1)!/2$  种不同的可能性可以尝试, 而且  $n!$  要远远大于  $2^n$  (见练习 5.5.1)。对于很大的  $n$ , 没有人知道能够保证解决这个问题的算法。

- 5.5.44 带回溯的欧几里得 TSP。修改你在练习 5.5.43 中的程序, 使得程序会停止搜索任何比当前已知最小旅行路径更长的路径。在小于 100 万的坐标系中随机取正整数值作为不同的  $n$ , 求两种情况下可能性的概率并作图表展示你的数据。

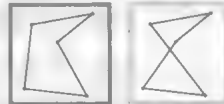
问题:  
在 5 个给定的城市中找到一个  
小于或等于 5 英里旅行的方法



所有可能性



解决方案



解决方案



一个欧几里得 TSP 实例

869

870  
871

## 构建一台计算机

本章的目标是向你展示我们日常使用的计算机的原理是多么简单。本章将假想一个简单的计算机，并详细描述它的构造过程。需要说明的是，许多在我们日常使用的计算设备的核心处理器所具有的特点，这个假想的计算机也具有。

你可能会很惊奇地发现，许多机器拥有相同的属性，即使是开发出的第一台计算机也是如此。因此，我们选择从历史的角度来讲解这一点。想象在一个没有计算机的世界，你会倍加渴望哪些设备，你的结论必然是与我们现在所拥有的计算机没什么差别！我们将从科学计算的角度讲述这个传奇的过程，其实从商业计算的观点来看，这点同样具有吸引力。

接下来，我们的目标是向你阐述：在一台简单机器上如何使用它自己的机器语言（machine language）实现 Java 编程中涉及的基本概念和结构。这个任务并不困难。我们还将详细介绍条件、循环、函数、数组以及链接结构。由于在 Java 中有相同的基本工具，因此，接下来会介绍几个在本书第一部分讨论过的计算任务，这些任务在较低的层面上也不难解决。

这个简单的假想计算机介于计算机与实际硬件电路之间。实际上硬件电路是通过改变状态以反映程序的动作。我们在下一章中会学习这些电路的工作原理。

当然了，这只是整个计算机的复杂工作原理中的一部分。最后，我们会以一个深刻的思考结束本章：我们可以使用一台机器来模拟另一台机器的操作。因此，我们可以轻松研究假想计算机，我们也可以开发未来将要建成的新机器，甚至可以研究那些可能永远不会构建的机器。

872  
873

## 6.1 信息表示

理解计算机工作原理的第一步是理解计算机内的信息表示方式。正如从 Java 编程中所了解到的，无论是数字、文本、可执行文件、图像、音频还是视频，凡是可以表示成 0 和 1 序列的所有形式的信息，都可以用数字计算机进行处理。对于每种类型的数据，标准编码方法已得到广泛应用：ASCII 标准将 128 个不同字符与 7 位二进制数相关联，MP3 文件格式严格地规定了将每个原始音频文件编码为 0/1 序列的方式，.png 图片格式指定了将数字图像中的像素最终表示为 0/1 序列的方法，等等。

在计算机中，信息通常组织为字（word）的形式，字是一个固定长度（称为字长）的位序列。后续你将了解到，字长在任何一台计算机的架构中都起着关键的作用。在早期的计算机中，典型的字长是 12 位或 16 位；32 位的字长又被广泛使用多年；而现在，64 位字长成为常态。

每台计算机中的信息内容是一个字的序列，每个字都是由固定数量的位（bit）组成，每位都是 0 或者 1。由于我们可以将每个字解释为二进制表示的数字，所以所有信息都是数字，所有数字也都是信息。

计算机内一个给定的位序列的含义取决于上下文。这是我们将在本章通篇都会重复的另一句话。例如，如你所见，根据上下文，我们可能会将二进制字符串 1111101011001110 解释为正整数 64 206、负整数 -1330、实数 -55 744.0 或两个字符的字符串“eN”。

二进制数字系统对计算机而言可能很方便，但对人类而言极其不便。如果你对这一事



实表示难以接受，你可以尝试记住 16 位二进制数 1111101011001110，然后合上本书并将其默写下来。为了适应计算机以二进制进行通信的需求，同时适应我们使用更紧凑的表示的需要，我们将在本节介绍十六进制（基数为 16）数字系统，它可以很方便地将二进制数字进行缩写。因此，我们从详细介绍十六进制开始。

[874]

**二进制和十六进制** 现在，考虑非负整数或者自然数（natural number），这是计数的基本数学抽象概念。自从巴比伦时代以来，人们已经用按位计数法（positional notation）和一个固定的基数（base）来表示整数。这些系统中最熟悉的是十进制，其基数为 10，每个正整数表示为 0 到 9 之间的数字串。具体来说， $d_n d_{n-1} \cdots d_2 d_1 d_0$  表示整数：

$$d_n 10^n + d_{n-1} 10^{n-1} + \cdots + d_2 10^2 + d_1 10^1 + d_0 10^0$$

例如，10 345 表示整数

$$10\ 345 = 1 \cdot 10\ 000 + 0 \cdot 1\ 000 + 3 \cdot 100 + 4 \cdot 10 + 5 \cdot 1$$

用任何一个比 1 大的整数替换基数 10，就可以得到一个不同的数字系统，用一串数字可以表示任何整数，其中数串中的每一位数字均在 0 到比基数小 1 的数字之间。在本章中，我们特别感兴趣的是二进制（基数为 2）和十六进制（基数为 16）。

**二进制。**当基数为 2 时，可以用一个 0/1 序列表示一个整数。在这种情况下，将每个二进制（基数为 2）数字——0 或 1——作为一位，这也是计算机中信息表示的基础。在这种情况下，这些位都是以 2 为底的幂的系数。具体地，位序列  $b_n b_{n-1} \cdots b_2 b_1 b_0$  表示整数：

$$b_n 2^n + b_{n-1} 2^{n-1} + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

例如，1100011 表示整数

$$99 = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1$$

用这个系统所能表示的  $n$  位数字的最大整数为  $2^n - 1$ ，此时  $n$  位均为 1。例如 8 位时，11111111 表示：

$$2^8 - 1 = 255 = 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1$$

表述这个限制的另一种说法是， $n$  位二进制数只能表示  $2^n$  个非负整数（ $0 \sim 2^n - 1$ ）。在使用计算机处理整数时，通常需要注意这些限制。另外，使用二进制符号的一个大的缺点是用二进制表示一个数字所需的位数比用十进制表示相同数字所需的位数要大得多。仅仅使用二进制与计算机进行通信会非常笨拙且不切实际。

[875]

**十六进制。**在十六进制中，十六进制数字序列  $h_n h_{n-1} \cdots h_2 h_1 h_0$  表示数字：

$$h_n 16^n + h_{n-1} 16^{n-1} + \cdots + h_2 16^2 + h_1 16^1 + h_0 16^0$$

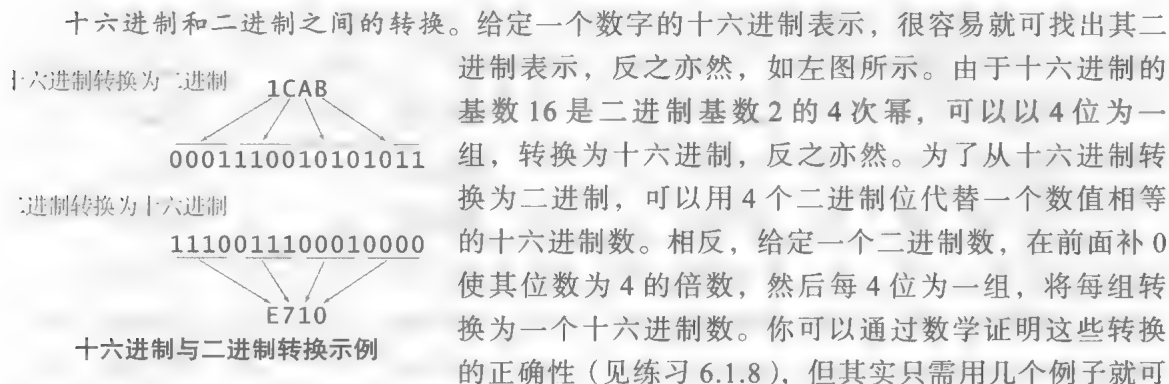
我们遇到的第一个复杂的问题是，由于基数是 16，就需要 0~15 之间的每一个数字。每个数字都需要用一个字符来表示，所以使用 A 来表示 10、B 表示 11、C 表示 12 等，如右表所示。例如，FACE 表示整数：

$$64\ 206 = 15 \cdot 16^3 + 10 \cdot 16^2 + 12 \cdot 16^1 + 14 \cdot 16^0$$

该数字与前文的 16 位二进制表示的数字相同。从这个例子中可以看出，以十六进制表示整数所需的十六进制数字的数量比用二进制表示相同整数所需位数要小得多（约四分之一）。此外，数字的多样性使得一个数更容易记忆。遇到 1111101011001110 你可能会迷茫，但你可以很容易就记住 FACE。

| 十进制 | 二进制  | 十六进制 |
|-----|------|------|
| 0   | 0000 | 0    |
| 1   | 0001 | 1    |
| 2   | 0010 | 2    |
| 3   | 0011 | 3    |
| 4   | 0100 | 4    |
| 5   | 0101 | 5    |
| 6   | 0110 | 6    |
| 7   | 0111 | 7    |
| 8   | 1000 | 8    |
| 9   | 1001 | 9    |
| 10  | 1010 | A    |
| 11  | 1011 | B    |
| 12  | 1100 | C    |
| 13  | 1101 | D    |
| 14  | 1110 | E    |
| 15  | 1111 | F    |

0~15 之间所有整数的表示



876

```
public static String toBinaryString(int n)
{
 if (n == 0) return "";
 if (n % 2 == 0)
 return toBinaryString(n/2) + '0';
 else
 return toBinaryString(n/2) + '1';
}
```

这是一种递归的方法，基于如下想法：数字的最后一个二进制字符可以表示为  $n\%2$ （如果  $n\%2$  为 0，则数字为 '0'；如果  $n\%2$  为 1，则数字为 '1'），字符串的剩余部分为  $n/2$  的二进制字符串表示。该程序的示例过程如右图所示。这种方法可以推广到十六进制的处理（或者其他进制）。另外，我们也关注将字符串表示转换为 Java 数据类型值。在下一节中，我们将学习一个完成此类转换的程序。

```
toBinaryString(109)
 toBinaryString(54)
 toBinaryString(27)
 toBinaryString(13)
 toBinaryString(6)
 toBinaryString(3)
 toBinaryString(1)
 toBinaryString(0)
 return ""
 return "1"
 return "11"
 return "110"
 return "1101"
 return "11011"
 return "110110"
 return "1101101"
```

toBinaryString(109) 的调用过程

当讨论计算机内发生的事情时，我们通常使用的语言是十六进制。如果需要， $n$  位计算机的一个字可以用  $n/4$  个十六进制位来表示，可立即转换为二进制。你可能在日常生活中已经观察到十六进制的使用。例如，当在网络上注册新设备时，需要知道设备的 MAC 地址。如 1a:ed:b1:b9:96:5e 可能就是一个 MAC 地址，它是用于标识设备的 48 位二进制数的十六进制缩写（使用一些多余的冒号和小写 a~f，而非我们使用的大写 A~F）。

877

| 十进制 | 二进制      | 十六进制 | 十进制 | 二进制      | 十六进制 | 十进制 | 二进制      | 十六进制 | 十进制 | 二进制      | 十六进制 |
|-----|----------|------|-----|----------|------|-----|----------|------|-----|----------|------|
| 0   | 00000000 | 00   | 32  | 00100000 | 20   | 64  | 01000000 | 40   | 96  | 01100000 | 60   |
| 1   | 00000001 | 01   | 33  | 00100001 | 21   | 65  | 01000001 | 41   | 97  | 01100001 | 61   |
| 2   | 00000010 | 02   | 34  | 00100010 | 22   | 66  | 01000010 | 42   | 98  | 01100010 | 62   |
| 3   | 00000011 | 03   | 35  | 00100011 | 23   | 67  | 01000011 | 43   | 99  | 01100011 | 63   |
| 4   | 00000100 | 04   | 36  | 00100100 | 24   | 68  | 01000100 | 44   | 100 | 01100100 | 64   |
| 5   | 00000101 | 05   | 37  | 00100101 | 25   | 69  | 01000101 | 45   | 101 | 01100101 | 65   |
| 6   | 00000110 | 06   | 38  | 00100110 | 26   | 70  | 01000110 | 46   | 102 | 01100110 | 66   |
| 7   | 00000111 | 07   | 39  | 00100111 | 27   | 71  | 01000111 | 47   | 103 | 01100111 | 67   |
| 8   | 00001000 | 08   | 40  | 00101000 | 28   | 72  | 01001000 | 48   | 104 | 01101000 | 68   |
| 9   | 00001001 | 09   | 41  | 00101001 | 29   | 73  | 01001001 | 49   | 105 | 01101001 | 69   |
| 10  | 00001010 | 0A   | 42  | 00101010 | 2A   | 74  | 01001010 | 4A   | 106 | 01101010 | 6A   |
| 11  | 00001011 | 0B   | 43  | 00101011 | 2B   | 75  | 01001011 | 4B   | 107 | 01101011 | 6B   |
| 12  | 00001100 | 0C   | 44  | 00101100 | 2C   | 76  | 01001100 | 4C   | 108 | 01101100 | 6C   |
| 13  | 00001101 | 0D   | 45  | 00101101 | 2D   | 77  | 01001101 | 4D   | 109 | 01101101 | 6D   |
| 14  | 00001110 | 0E   | 46  | 00101110 | 2E   | 78  | 01001110 | 4E   | 110 | 01101110 | 6E   |
| 15  | 00001111 | 0F   | 47  | 00101111 | 2F   | 79  | 01001111 | 4F   | 111 | 01101111 | 6F   |
| 16  | 00010000 | 10   | 48  | 00110000 | 30   | 80  | 01010000 | 50   | 112 | 01110000 | 70   |
| 17  | 00010001 | 11   | 49  | 00110001 | 31   | 81  | 01010001 | 51   | 113 | 01110001 | 71   |
| 18  | 00010010 | 12   | 50  | 00110010 | 32   | 82  | 01010010 | 52   | 114 | 01110010 | 72   |
| 19  | 00010011 | 13   | 51  | 00110011 | 33   | 83  | 01010011 | 53   | 115 | 01110011 | 73   |
| 20  | 00010100 | 14   | 52  | 00110100 | 34   | 84  | 01010100 | 54   | 116 | 01110100 | 74   |
| 21  | 00010101 | 15   | 53  | 00110101 | 35   | 85  | 01010101 | 55   | 117 | 01110101 | 75   |
| 22  | 00010110 | 16   | 54  | 00110110 | 36   | 86  | 01010110 | 56   | 118 | 01110110 | 76   |
| 23  | 00010111 | 17   | 55  | 00110111 | 37   | 87  | 01010111 | 57   | 119 | 01110111 | 77   |
| 24  | 00011000 | 18   | 56  | 00111000 | 38   | 88  | 01011000 | 58   | 120 | 01111000 | 78   |
| 25  | 00011001 | 19   | 57  | 00111001 | 39   | 89  | 01011001 | 59   | 121 | 01111001 | 79   |
| 26  | 00011010 | 1A   | 58  | 00111010 | 3A   | 90  | 01011010 | 5A   | 122 | 01111010 | 7A   |
| 27  | 00011011 | 1B   | 59  | 00111011 | 3B   | 91  | 01011011 | 5B   | 123 | 01111011 | 7B   |
| 28  | 00011100 | 1C   | 60  | 00111100 | 3C   | 92  | 01011100 | 5C   | 124 | 01111100 | 7C   |
| 29  | 00011101 | 1D   | 61  | 00111101 | 3D   | 93  | 01011101 | 5D   | 125 | 01111101 | 7D   |
| 30  | 00011110 | 1E   | 62  | 00111110 | 3E   | 94  | 01011110 | 5E   | 126 | 01111110 | 7E   |
| 31  | 00011111 | 1F   | 63  | 00111111 | 3F   | 95  | 01011111 | 5F   | 127 | 01111111 | 7F   |

0~127 之间整数的十进制、8 位二进制和 2 位十六进制表示

| 十进制 | 二进制      | 十六进制 | 十进制 | 二进制      | 十六进制 | 十进制 | 二进制      | 十六进制 | 十进制 | 二进制      | 十六进制 |
|-----|----------|------|-----|----------|------|-----|----------|------|-----|----------|------|
| 128 | 10000000 | 80   | 160 | 10100000 | A0   | 192 | 11000000 | C0   | 224 | 11100000 | E0   |
| 129 | 10000001 | 81   | 161 | 10100001 | A1   | 193 | 11000001 | C1   | 225 | 11100001 | E1   |
| 130 | 10000010 | 82   | 162 | 10100010 | A2   | 194 | 11000010 | C2   | 226 | 11100010 | E2   |
| 131 | 10000011 | 83   | 163 | 10100011 | A3   | 195 | 11000011 | C3   | 227 | 11100011 | E3   |
| 132 | 10000100 | 84   | 164 | 10100100 | A4   | 196 | 11000100 | C4   | 228 | 11100100 | E4   |
| 133 | 10000101 | 85   | 165 | 10100101 | A5   | 197 | 11000101 | C5   | 229 | 11100101 | E5   |
| 134 | 10000110 | 86   | 166 | 10100110 | A6   | 198 | 11000110 | C6   | 230 | 11100110 | E6   |
| 135 | 10000111 | 87   | 167 | 10100111 | A7   | 199 | 11000111 | C7   | 231 | 11100111 | E7   |
| 136 | 10001000 | 88   | 168 | 10101000 | A8   | 200 | 11001000 | C8   | 232 | 11101000 | E8   |
| 137 | 10001001 | 89   | 169 | 10101001 | A9   | 201 | 11001001 | C9   | 233 | 11101001 | E9   |
| 138 | 10001010 | 8A   | 170 | 10101010 | AA   | 202 | 11001010 | CA   | 234 | 11101010 | EA   |
| 139 | 10001011 | 8B   | 171 | 10101011 | AB   | 203 | 11001011 | CB   | 235 | 11101011 | EB   |
| 140 | 10001100 | 8C   | 172 | 10101100 | AC   | 204 | 11001100 | CC   | 236 | 11101100 | EC   |
| 141 | 10001101 | 8D   | 173 | 10101101 | AD   | 205 | 11001101 | CD   | 237 | 11101101 | ED   |
| 142 | 10001110 | 8E   | 174 | 10101110 | AE   | 206 | 11001110 | CE   | 238 | 11101110 | EE   |
| 143 | 10001111 | 8F   | 175 | 10101111 | AF   | 207 | 11001111 | CF   | 239 | 11101111 | EF   |
| 144 | 10010000 | 90   | 176 | 10110000 | B0   | 208 | 11010000 | D0   | 240 | 11110000 | F0   |
| 145 | 10010001 | 91   | 177 | 10110001 | B1   | 209 | 11010001 | D1   | 241 | 11110001 | F1   |
| 146 | 10010010 | 92   | 178 | 10110010 | B2   | 210 | 11010010 | D2   | 242 | 11110010 | F2   |
| 147 | 10010011 | 93   | 179 | 10110011 | B3   | 211 | 11010011 | D3   | 243 | 11110011 | F3   |
| 148 | 10010100 | 94   | 180 | 10110100 | B4   | 212 | 11010100 | D4   | 244 | 11110100 | F4   |
| 149 | 10010101 | 95   | 181 | 10110101 | B5   | 213 | 11010101 | D5   | 245 | 11110101 | F5   |
| 150 | 10010110 | 96   | 182 | 10110110 | B6   | 214 | 11010110 | D6   | 246 | 11110110 | F6   |
| 151 | 10010111 | 97   | 183 | 10110111 | B7   | 215 | 11010111 | D7   | 247 | 11110111 | F7   |
| 152 | 10011000 | 98   | 184 | 10111000 | B8   | 216 | 11011000 | D8   | 248 | 11111000 | F8   |
| 153 | 10011001 | 99   | 185 | 10111001 | B9   | 217 | 11011001 | D9   | 249 | 11111001 | F9   |
| 154 | 10011010 | 9A   | 186 | 10111010 | BA   | 218 | 11011010 | DA   | 250 | 11111010 | FA   |
| 155 | 10011011 | 9B   | 187 | 10111011 | BB   | 219 | 11011011 | DB   | 251 | 11111011 | FB   |
| 156 | 10011100 | 9C   | 188 | 10111100 | BC   | 220 | 11011100 | DC   | 252 | 11111100 | FC   |
| 157 | 10011101 | 9D   | 189 | 10111101 | BD   | 221 | 11011101 | DD   | 253 | 11111101 | FD   |
| 158 | 10011110 | 9E   | 190 | 10111110 | BE   | 222 | 11011110 | DE   | 254 | 11111110 | FE   |
| 159 | 10011111 | 9F   | 191 | 10111111 | BF   | 223 | 11011111 | DF   | 255 | 11111111 | FF   |

128~255 之间整数的十进制、8 位二进制和 2 位十六进制表示

本章的后续部分将主要关注小于 256 的整数，这些整数可以用 8 位二进制或 2 位十六进制来表示。前文中使用一张表列出了 0 到 255 的所有整数的十进制、二进制和十六进制表示，以作为参考。花费几分钟学习该表是值得的，它能够让你更有信心地使用这些整数，并了解这些表示形式之间的关系。如果你已经熟悉到认为这个表格是在浪费篇幅，那么我们的目标就已经达到了！

**解析和字符串表示** 在整数的不同表示方法之间转换是一项非常有趣的计算任务，我们在程序 1.3.7 中首先进行了探讨，之后在练习 2.3.15 中重新进行了讨论。我们也一直在使用 Java 的这种方法。接下来，为了巩固各种基数的按位计数表示法，我们来探讨将任何数字从一个基数转换为另一个基数的程序。

**解析。**将字符串转换为内部表示形式称为解析（parsing）。从 1.1 节开始，我们一直在使用像 `Integer.parseInt()` 这样的 Java 方法以及类似 `StdIn.readInt()` 这样我们自己的方法，将输入的字符串中的数字转换为 Java 数据类型的值。我们一直在使用十进制数（使用 0 到 9 之间的字符串表示），现在我们来查看一个方法，用以解析在任何基数下的数字。为简单起见，将其限制在不超过 36 的基数上，并扩展十六进制，约定使用字母 A 到 Z 表示从 10 到 35 的数字。注意：Java 的 `Integer` 类有一个双参数的 `parseInt()` 方法，它具有类似的功能，除此之外，它也能处理负整数。

| i | n   | 看到的字符   |
|---|-----|---------|
| 0 | 1   | 1       |
| 1 | 3   | 11      |
| 2 | 6   | 110     |
| 3 | 13  | 1101    |
| 4 | 27  | 11011   |
| 5 | 54  | 110110  |
| 6 | 109 | 1101101 |

`parseInt(1101101,2)` 的过程

现代数据类型的特征之一是内部表示被隐藏，只能使用数据类型值上定义的操作来处理数据。具体来说，最好是限制对表示数据类型值的位的直接引用，而是使用数据类型操作来代替（这样可以有效避免不同处理器硬件在底层实现上的差异——译者注）。

解析数字所需要进行的第一个原语操作是将字符转换为整数的方法。练习 6.1.12 给出了一个 `toInt()` 方法，将 0~9 或 A~Z 范围内的字符作为参数，返回一个 0~35 之间的 `int` 值（数字对应 0~9，字母对应 10~35）。有了这个原语，程序 6.1.1 中很简单就实现了一个方法 `parseInt()`，可以解析以任意值 *b*（从 2 到 36 之间的整数）为基数的字符串的整数表示，输入参数为整数的字符串表示，并返回该整数在 Java 中的 `int` 值。通常，我们通过一段循环代码来计算这个数字。每次循环时，`int` 值 *n* 用于存储已经处理的所有字符所对应的整数。在每一次循环中，*n* 需要乘以基数，再加上下一个数字的值。右边的例子详细展示了这个过程：每一轮循环中 *n* 的值总是基数乘以 *n* 的前一个值加上下一个数字（用粗体标出）。为了解析 1101101，要计算  $0 \cdot 2 + 1 = 1$ ， $1 \cdot 2 + 1 = 3$ ， $3 \cdot 2 + 0 = 6$ ， $6 \cdot 2 + 1 = 13$ ， $13 \cdot 2 + 1 = 27$ ， $27 \cdot 2 + 0 = 54$  以及  $54 \cdot 2 + 1 = 109$ 。为了解析十六进制数 FACE，需要计算出  $0 \cdot 16 + 15 = 15$ ， $15 \cdot 16 + 10 = 250$ ， $250 \cdot 16 + 12 = 4\,012$  和  $4\,012 \cdot 16 + 14 = 64\,206$ 。这是通过霍纳法则（Horner's method）（霍纳法则在中国被称为秦九韶算法。——译者注）进行多项式评估的一种特殊情况（见练习 2.1.31）。

[880] 解析数字所需要进行的第一个原语操作是将字符转换为整数的方法。练习 6.1.12 给出了一个 `toInt()` 方法，将 0~9 或 A~Z 范围内的字符作为参数，返回一个 0~35 之间的 `int` 值（数字对应 0~9，字母对应 10~35）。有了这个原语，程序 6.1.1 中很简单就实现了一个方法 `parseInt()`，可以解析以任意值 *b*（从 2 到 36 之间的整数）为基数的字符串的整数表示，输入参数为整数的字符串表示，并返回该整数在 Java 中的 `int` 值。通常，我们通过一段循环代码来计算这个数字。每次循环时，`int` 值 *n* 用于存储已经处理的所有字符所对应的整数。在每一次循环中，*n* 需要乘以基数，再加上下一个数字的值。右边的例子详细展示了这个过程：每一轮循环中 *n* 的值总是基数乘以 *n* 的前一个值加上下一个数字（用粗体标出）。为了解析 1101101，要计算  $0 \cdot 2 + 1 = 1$ ， $1 \cdot 2 + 1 = 3$ ， $3 \cdot 2 + 0 = 6$ ， $6 \cdot 2 + 1 = 13$ ， $13 \cdot 2 + 1 = 27$ ， $27 \cdot 2 + 0 = 54$  以及  $54 \cdot 2 + 1 = 109$ 。为了解析十六进制数 FACE，需要计算出  $0 \cdot 16 + 15 = 15$ ， $15 \cdot 16 + 10 = 250$ ， $250 \cdot 16 + 12 = 4\,012$  和  $4\,012 \cdot 16 + 14 = 64\,206$ 。这是通过霍纳法则（Horner's method）（霍纳法则在中国被称为秦九韶算法。——译者注）进行多项式评估的一种特殊情况（见练习 2.1.31）。

| i | n     | 看到的字符  |
|---|-------|--------|
| 0 | 15    | "F"    |
| 1 | 250   | "FA"   |
| 2 | 4012  | "FAC"  |
| 3 | 64206 | "FACE" |

`parseInt(FACE,16)` 的过程

程序6.1.1 将一个自然数从一种进制转换为另一种进制

```

public class Convert
{
 public static int toInt(char c)
 { /* 见练习6.1.12 */ }
 public static char toChar(int i)
 { /* 见练习6.1.13 */ }
 public static int parseInt(String s, int d)
 {
 int n = 0;
 for (int i = 0; i < s.length(); i++)
 n = d*n + toInt(s.charAt(i));
 return n;
 }
 public static String toString(int n, int d)
 {
 if (n == 0) return "";
 return toString(n/d, d) + toChar(n % d);
 }
 public static void main(String[] args)
 {
 while (!StdIn.isEmpty())
 {
 String s = StdIn.readString();
 int baseFrom = StdIn.readInt();
 int baseTo = StdIn.readInt();
 int n = parseInt(s, baseFrom);
 StdOut.println(toString(n, baseTo));
 }
 }
}

```

```

% java Convert
1101101 2 10
109
FACE 16 10
64206
FACE 16 2
1111101011001110
109 10 2
1101101
64206 10 16
FACE
64206 10 32
1UME
1UME 32 10
64206

```

该通用转换程序从标准输入中读取字符串和两个基数，使用parseInt()和toString()将整数在第一个基数下的表示转换为相同整数在另一个基数下的表示

为简单起见，在这段代码中我们并未包含错误检查。例如，如果由toInt()返回的值不小于基数，则parseInt()应该抛出异常。此外，发生溢出时也应该抛出异常，因为输入的字符串表示的数字可能大于Java中int可以表示的数字。

字符串表示。使用toString()方法计算一个数据类型值的字符串表示也是本书开始以来一直在做的事情。我们可以通过扩展本节前面讲到的十进制转二进制的方法来实现这一功能(练习2.3.15的解决方案)，也是一种类似的递归方法。用代码实现计算任何给定基数下整数的字符串表示形式的方法会是一件很有意思的事情。需要说明的是，Java的Integer类有一个双参数的toString()方法可以实现类似功能。

同样，所需的第一个原语操作是将一个整数转换为字符(数字)的方法。练习6.1.13给出了一个toChar()方法，参数为0~35之间的一个int值，返回值为0~9(对于小于10的值)或A~Z(对于10~35之间的值)之间的一个字符。有了这个原语，程序6.1.1中的toString()方法甚至可以比parseInt()更简单。这是一个递归方法，基本思想是最后一个数字是n%d的字符表示，而其余的字符串是n/d的字符串表示。这种计算本质上是解析计算的逆向计算，正如以下所示的调用跟踪中看到的那样。

```

toString(64206, 16)
 toString(4012, 16)
 toString(250, 16)
 toString(15, 16)
 toString(0, 16)
 return ""
 return "F"
 return "FA"
 return "FAC"
 return "FACE"

```

toString(64206,16)的调用过程

881  
}  
882

在讨论计算机中的字时,需要为它加上前导0,这样就可以更加方便地描述所有的位。出于这个理由,在 Convert 中包含一个三参数版本的 toString(), 其中第三个参数是返回的字符串中要求的位数。例如,调用 toString(15,16,4) 返回 000F, 调用 toString(14,2,16) 返回 0000000000001110。该版本的实现留作练习(见练习 6.1.15)。

程序 6.1.1 将所有这些想法融合到一起,是一个计算两种进制之间数字的通用工具。当标准输入流非空时,测试客户端的主循环从标准输入中读取字符串,后跟两个整数(当前字符串表示的基数以及结果表示的基数),执行指定的转换并输出结果。为了完成这项任务,它使用 parseInt() 将输入字符串转换为一个 Java int 型数据,然后使用 toString() 将该 Java int 型数据转换为特定基数下数字的字符串表示。我们鼓励你下载并使用此工具来熟悉数字转换和表示。

883

**整数算术** 要考虑的对整数的第一个操作是基本算术操作,如加法和乘法。实际上,早期计算设备的最初目的就是重复执行这样的操作。在下一章中,我们将学习如何构建这样的计算设备,因为每台计算机都有执行这样操作的内置硬件。此时,在小学时期学到的十进制的基本运算对于二进制和十六进制同样适用。

**加法。**在小学时期已经学习过两个十进制数字的加法,规则是将最低位的两个数字(最右边的数字)相加;如果和大于10,则向前位进1,该位为和模10的余数。下一位数字重复该过程,但要注意需要加上进位。相同的程序可以扩展到任一进制。例如,如果使用十六进制,两个加数为7和E,那么该位为5,并向前进位1,因为7+E得到十六进制的15。如果使用二进制,两个加数都为1,并且再加一个1,则该位为1,并且向前位进1,因此1+1+1得二进制的11。左边的例子为  $4567_{10} + 366_{10} = 4933_{10}$  的十进制、十六进制和二进制的计算过程。同在小学时期学到的一样,我们会省略前导的零。

**无符号整数。**如果要用一个计算机字表示整数,那么可以表示的整数的数量和大小会存在限制。如上所述,用一个  $n$  位的字,只能表示  $2^n$  个整数。如果只需要非负(或无符号)整数,通常

| 位数 | 最小值 | 最大值                        |
|----|-----|----------------------------|
| 4  | 0   | 15                         |
| 16 | 0   | 65 535                     |
| 32 | 0   | 4 294 967 295              |
| 64 | 0   | 18 446 744 073 709 551 615 |

能表示的无符号整数

的选择是使用带有前导零的二进制表示整数0到  $2^n-1$ , 这样每个字都对应一个整数,在定义范围内的每一个整数都对应一个字。右表所示为一个4位的字所能表示的16个无符号数,左表所示为典型的计算机中使用的16位、32位、64位的字所能表示

| 十进制 | 二进制  |
|-----|------|
| 0   | 0000 |
| 1   | 0001 |
| 2   | 0010 |
| 3   | 0011 |
| 4   | 0100 |
| 5   | 0101 |
| 6   | 0110 |
| 7   | 0111 |
| 8   | 1000 |
| 9   | 1001 |
| 10  | 1010 |
| 11  | 1011 |
| 12  | 1100 |
| 13  | 1101 |
| 14  | 1110 |
| 15  | 1111 |

4位整数(无符号)

884 的整数的范围。

**溢出。**如在1.2节中看到的Java编程一样,需要注意确保算术运算结果不超过能够存储的最大值。超过最大值的情况被称为溢出(overflow)。对于无符号整数的加法,溢出很容



易检测：如果最后一次（最左边的位）加法产生了进位，则结果太大，将无法表示。即使是在计算机硬件中（后面将会见到），检查一个数位的值也很容易，所以计算机和编程语言通常都提供测试这种可能性的低级指令。值得注意的是，Java 并没有提供这样的功能（参见 1.2 节中的问答环节）。

乘法。类似地，如右图所示，小学学到的乘法算法也完美适用于任何进制（二进制的例子很难跟踪，因为会产生大量的级联进位：如果你真想试试看，你可以做一次加 2 操作）。实际上，计算机科学家已经发现了一种乘法算法，相比之下，这种算法更适合在计算机中实现，并且更加有效。在早期的计算机中，程序员不得不在软件中进行乘法运算（稍后在练习 6.3.35 中会演示这种实现方式）。要注意，相比加法运算，开发乘法算法时溢出是一个更大的问题，因为结果中的位数可能达到操作数位数的两倍。也就是说，当两个  $n$  位的数字相乘时，需要为结果准备  $2n$  位。

本书无法深入描述使用计算机硬件执行算术运算的所有技术。当然，你希望计算机能有效地执行除法、求幂及其他操作，而我们计划详细说明加 / 减法，简要介绍其他操作。

除无符号整数外，你也希望能够用负数和实数进行计算。接下来将简要描述为了支持这些操作而使用的标准表示方法。

负数 计算机设计人员早期发现，通过使用称为二进制补码（two's complement）的表示法，修改整数数据类型使其可以支持负数并不是很困难。

我们可能会想到的第一种方法就是符号加数值（sign-and-magnitude）表示法，其中第一位表示符号，剩余位表示数字的值。例如，使用 4 位表示时，0101 表示 +5，1101 表示 -5。相比之下，在一个  $n$  位的二进制补码中，正数的表示方法与之前相同，负数  $-x$  使用二进制数  $2^n - x$  表示。例如，右表所示为一个 4 位的字使用二进制补码所能表示的 16 个整数。可以看出 0101 仍然表示 +5，但是 1011 表示 -5，因为  $2^4 - 5 = 1110 = 1011_2$ 。

由于保留了一个符号位，所以使用二进制补码所能表示的整数数量约为使用相同比特数所能表示的无符号整数数量的一半。从 4 位示例中可以看出，二进制补码中存在轻微的不对称：4 位可以表示 1~7 的正数、-8~-1 的负数以及 0。一般来说，在  $n$  位二进制补码中，能够表示的最小负数为  $-2^{n-1}$ ，最大正数为  $2^{n-1} - 1$ 。下表所示为 16 位二进制补码所能表示的最小和（绝对值）最大整数值。

进位表示  
有溢出  
↓  
1000000111111000  
1111111111111000  
0000000000001000  
0000000000000000  
溢出（16 位无符号数）

十进制  
4567  
\* 366  
27402  
27402  
13701  
1671522

十六进制  
11D7  
\* 16E  
F9C2  
6B0A  
11D7  
198162

二进制  
1000111010111  
\* 0000101101110  
1000111010111  
1000111010111  
1000111010111  
1000111010111  
1000111010111  
1000111010111  
110011000000101100010  
乘法示例

885

| 十进制 | 二进制  |
|-----|------|
| 0   | 0000 |
| 1   | 0001 |
| 2   | 0010 |
| 3   | 0011 |
| 4   | 0100 |
| 5   | 0101 |
| 6   | 0110 |
| 7   | 0111 |
| -8  | 1000 |
| -7  | 1001 |
| -6  | 1010 |
| -5  | 1011 |
| -4  | 1100 |
| -3  | 1101 |
| -2  | 1110 |
| -1  | 1111 |

4 位整数（二进制补码）

| 十进制    | 二进制              |
|--------|------------------|
| 0      | 0000000000000000 |
| 1      | 0000000000000001 |
| 2      | 0000000000000010 |
| 3      | 0000000000000011 |
| ...    | ...              |
| 32765  | 0111111111111101 |
| 32766  | 0111111111111110 |
| 32767  | 0111111111111111 |
| -32768 | 1000000000000000 |
| -32767 | 1000000000000001 |
| -32766 | 1000000000000010 |
| -32765 | 1000000000000011 |
| ...    | ...              |
| -3     | 1111111111111101 |
| -2     | 1111111111111110 |
| -1     | 1111111111111111 |

16 位整数 (二进制补码)

二进制补码能够胜过符号加数值表示法成为标准有两个主要原因。第一, 因为 0 只有一种表示方法 (全为 0 的二进制串), 所以检验是否为 0 会非常简单。第二, 算术操作会很容易实现——稍后本节将讨论这一点。另外, 在符号加数值表示法中, 前导位表示符号, 因此检验一个值是否为负会非常简单。事实上, 构建计算机硬件是一个非常困难的过程, 因此, 如果一个数字表示方法的规则能够简化这个过程, 那么这无疑是—

[886]

一个非常有意义的改进。

**加法。**将两个  $n$  位二进制补码整数相加也很容易: 使用无符号整数加法规则将其相加。例如,  $2+(-7)=0010+1001=1011=-5$ 。证明结果仍然在范围内 ( $-2^{n-1} \sim 2^{n-1}-1$ ) 并不困难:

|                    |      |
|--------------------|------|
| 0000000000000000   |      |
| 0000000001000000   | 64   |
| 0000000000101010   | +42  |
| 0000000001101010   | 106  |
| 1111111111100000   |      |
| 0000000001000000   | 64   |
| 1111111111010110   | -42  |
| 0000000000010110   | 22   |
| 0000000000000000   |      |
| 1111111111100000   | -64  |
| 0000000000101010   | +42  |
| 1111111111101010   | -22  |
| 1111111111100000   |      |
| 1111111111100000   | -64  |
| 1111111111101010   | -42  |
| 111111111110010110 | -106 |

加法 (16 位二进制补码)

- 如果两个整数都为非负, 则只要结果 (绝对值) 小于  $2^{n-1}$ , 标准二进制加法就仍然适用。
- 如果两个整数都为负, 则和为:  $(2^n - x) + (2^n - y) = 2^n + 2^n - (x + y)$ 。
- 如果  $x$  为负,  $y$  为正, 且结果为负, 则有:  $(2^n - x) + y = 2^n - (x - y)$ 。
- 如果  $x$  为负,  $y$  为正, 且结果为正, 则有:  $(2^n - x) + y = 2^n + (y - x)$ 。

在第二种和第四种情况下,  $2^n$  项不会对  $n$  位的结果产生影响 ( $2^n$  超出了  $n$  位二进制数的表示范围, 且低  $n$  位二进制数全是 0, 因此对低  $n$  位不产生任何影响——译者注), 所以标准二进制加法 (忽略进位) 可以得出正确的结果。溢出检测比无符号整数更加复杂, 我们将其留到问答环节。

```

00000000000001010 10
1111111111110101 翻转所有位
+0000000000000001 加1
1111111111110110 -10

```

```

1111111111110110 -10
0000000000001001 翻转所有位
+0000000000000001 加1
0000000000001010 10

```

```

0001001101001110 4942
1110110010110001 翻转所有位
+0000000000000001 加1
1110110010110010 -4942

```

用二进制补码求数字的相反数

二进制补码中，0111 加 1 得到 1000；在 16 位二进制补码中，0111111111111111 加 1 得到 1000000000000000（注意这是对二进制补码整数加 1 不会产生预期结果的唯一情形）。练习 1.2.10 中的其他情形也可以得到解释。几十年来，这样的行为让那些没有花费时间去了解二进制补码的程序员不知所措。下面是一个有说服力的例子：在 Java 中，调用 Math.abs(-2 147 483 648) 却返回 -2 147 483 648，一个负整数！

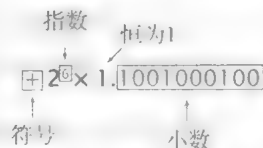
|     |     |                             |
|-----|-----|-----------------------------|
| 16位 | 最小值 | - 32 768                    |
|     | 最大值 | 32 767                      |
| 32位 | 最小值 | - 2 147 483 648             |
|     | 最大值 | 2 147 483 647               |
| 64位 | 最小值 | - 9 223 372 036 854 775 808 |
|     | 最大值 | 9 223 372 036 854 775 807   |

二进制补码所能表示的整数

**实数** 如何表示实数呢？这个任务比表示整数更有挑战，因为必须做出很多选择。在数字计算发展的头二十年，早期计算机设计者尝试了很多选择，演变出很多种不同的存储格式。实数运算很长时间内都只能在软件上实现，且与整数运算相比非常慢。

到 20 世纪 80 年代中期，对标准的需求越来越强烈（不同的计算机可能会在相同的计算中得到略微不同的结果），所以电气和电子工程师协会（IEEE）开发了 IEEE 754 标准，这个标准至今仍在发展中。该标准包含的内容非常广泛——你可能不会想了解它的全部细节——但我们可以在此简要描述其基本想法。我们将 16 位版本称作 IEEE 754 的半精度二进制浮点教格式，简称为 binary16。相同的基本思想同样适用于 Java 中使用的 32 位和 64 位版本。

**浮点数**：在计算机系统中常用的实数表示法是浮点数。它就像科学计数法一样，不同之处在于所有的数字都用二进制表示。在科学计数法中，我们习惯使用类似  $+6.022\ 141\ 3 \times 10^{23}$  的数字，这样的数字包括符号、系数和指数。通常，该数字被表示时系数为一个（非零）数字。这种表示方法称为标准化条件（normalization condition）（即要求系数是一个小于基数的数字。——译者注）。在浮点数中，同样也需要这三个元素。



剖析浮点数

**减法**。要计算  $x - y$ ，我们计算  $x + (-y)$ 。也就是说，如果知道如何计算  $-y$ ，就仍然可以使用二进制加法。结果表明，在二进制补码中求一个数字的相反数非常简单：取反，加 1。该过程的三个示例如左图所示——我们将证明留作练习。

Java 程序员也需要了解二进制补码的相关原理，因为 short、int 和 long 型值分别使用 16 位、32 位和 64 位的二进制补码来表示负整数。这也解释了 Java 程序员为什么必须知道这些类型值的界限（如下图所示）。887

另外，Java 的二进制补码表示方法解释了在 1.2 节中观察到的溢出现象（见 1.2 节问答环节及练习 1.2.10）。在该示例中可以看到，对于任意 Java 整数类型，最大正整数加 1 的结果是最大负整数。在 4 位

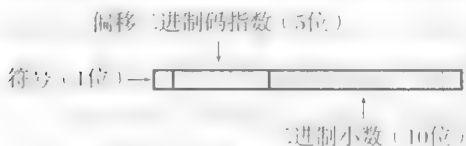
887

888

符号。浮点数的第一位是它的符号 (sign)。符号位为 0 时, 数字为正 (或零), 如果为 1 则为负。检查一个数字是正还是负很容易。

指数。浮点数接下来的  $t$  位用于表示其指数。binary16、binary32 和 binary64 表示指数的位数分别为 5 位、8 位、11 位。浮点数的指数不使用二进制补码表示, 而是使用偏移二进制码 (offset binary) 表示, 其中分别取  $t = 5、8、11$  时的  $R = 2^{t-1} - 1$  ( $t = 5、8、11$  分别对应 15、127、1023), 用  $x + R$  的二进制表示  $-R \sim R + 1$  之间的任意十进制整数  $x$ 。右表所示为  $-15 \sim +16$  之间所有整数的 5 位偏移二进制码表示。在标准中, 0000 和 1111 用于其他目的。

小数。浮点数中剩余位用于表示系数: binary16、binary32 和 binary64 分别用 10 位、23 位、53 位来表示。标准化条件意味着系数中小数位前的数字始终为 1, 因此我们不需要在表示中包含该数字!



IEEE 754 半精度浮点数格式

给定这些规则, 对以 IEEE 754 格式编码的数字进行解码的过程就变得很简单, 如下图所示的示例所示。根据标准, 给定 16 位中的第一位是符号, 接下来的五位是指数

(-610) 的偏移二进制编码, 接下来的 10 位是小数, 其定义了系数  $1.101_2$ 。由底部示例所示, 编码过程相对较为复杂, 这是由于需要标准化并扩展二进制转换使其包含小数。同样, 第一位是符号位, 接下来的 5 位是指数, 再接下来的 10 位是小数。这些任务即使在像 Java 这样的高级语言中也是一个具有挑战性的编程练习 (参见练习 6.1.25, 但首先阅读下一小节中的位操作), 所以你可以想象为什么早期计算机硬件不支持浮点数以及为什么要花这么长的时间来发展标准。

IEEE 754 转换为十进制

1010011010000000

$$-2^{9-15} \times 1.101_2 = -2^{-6} (1 + 2^{-1} + 2^{-3}) = -0.0253906250_{10}$$

十进制转换为 IEEE 754

$$100.25_{10} = 2^6 (1 + 2^{-1} + 2^{-4} + 2^{-8}) = +2^{21-15} \times 1.10010001_2$$

0101011001000100

浮点数与十进制相互转换的例子

Java 的 Float 和 Double 数据类型都包含 floatToIntBits() 方法, 可以使用这个方法检查浮点数编码。例如, 调用

Convert.toString(Float.floatToIntBits(100.25), 2, 16)

将输出结果: 0101011001000100, 与上面第二个例子的预期结果相同。

| 十进制 | 二进制   |
|-----|-------|
| -15 | 00000 |
| -14 | 00001 |
| -13 | 00010 |
| -12 | 00011 |
| -11 | 00100 |
| -10 | 00101 |
| -9  | 00110 |
| -8  | 00111 |
| -7  | 01000 |
| -6  | 01001 |
| -5  | 01010 |
| -4  | 01011 |
| -3  | 01100 |
| -2  | 01101 |
| -1  | 01110 |
| 0   | 01111 |
| 1   | 10000 |
| 2   | 10001 |
| 3   | 10010 |
| 4   | 10011 |
| 5   | 10100 |
| 6   | 10101 |
| 7   | 10110 |
| 8   | 10111 |
| 9   | 11000 |
| 10  | 11001 |
| 11  | 11010 |
| 12  | 11011 |
| 13  | 11100 |
| 14  | 11101 |
| 15  | 11110 |
| 16  | 11111 |

5 位整数 (偏移二进制码)

算术运算。对浮点数进行算术运算也可以作为有趣的编程练习。例如，两个浮点数相乘需要以下步骤：

- 确定符号。
- 将指数相加。
- 将小数相乘。
- 将结果标准化。

如果有兴趣，你可以研究以下该过程的细节，并通过完成练习 6.1.25 研究相应的加法和乘法过程。加法实际上比乘法复杂得多，因为第一步需要“非标准化”使得指数匹配。

使用浮点数进行计算通常具有挑战性，因为它们大部分都是我们真正想处理的实数的近似值，近似值的误差会在长期的计算过程中累积。由于 64 位格式（Java 中 double 数据类型所使用的）具有 32 位格式（Java 中 float 数据类型所使用的）两倍数量的位数，大多数程序员会选择使用 double 以减少近似误差的影响，本书中也是这么做的。

890

**用于操作二进制位的 Java 代码** 从实数的浮点数编码中可以看出，使用二进制编码信息可能会变得非常复杂。接下来，我们将介绍在 Java 中可以使用的工具，这些工具将有助于信息编解码程序的开发。Java 将整数值定义为二进制补码形式，并明确指出 short、int、long 数据类型的值分别为 16 位、32 位、64 位二进制补码，这是这些工具能够正常工作的基础。并非所有的语言都能这样做，有些语言将这类功能的代码留到更低级语言实现，有些语言为比特序列定义了一种明确的数据类型，或者可能需要进行困难或代价较高的转换。我们主要介绍关于 32 位 int 型值的操作，但这些操作对 short 和 long 型值也适用。

**二进制和十六进制常量（literal）。**在 Java 中，可以用二进制（需要前缀 0b）和十六进制（需要前缀 0x）指定整数型常量值。这是两种直接使用二进制编码声明常量值的方式。你可以在任何使用十进制常量的地方使用这样的常量，它仅仅是指定整数值的另一种方式。如果你给一个 int 型变量指定一个十六进制常量，并且位数少于 8，则 Java 将自动在前面补 0。下面展示了几个例子。

| 二进制常量                              | 十六进制常量     | 简写     |
|------------------------------------|------------|--------|
| 0b01000000010101000100111101011001 | 0x40544F59 |        |
| 0b11111111111111111111111111111111 | 0x0000000F | 0xF    |
| 0b00000000000000000001001000110100 | 0x00001234 | 0x1234 |
| 0b00000000000000001000101000101011 | 0x00008A2B | 0x8A2B |

**Java 代码中的移位和按位运算。**为了允许客户端能够针对一个 int 型变量的位进行操作，Java 支持下面所列出的类似操作。我们可以进行取反（将 0 变为 1，将 1 变为 0）、按位逻辑操作（应用后面定义的比特对的 and、or、xor 函数）、左移或右移一定数量的位。对于右移，有两种选择：逻辑移位（logical shift），其中用 0 在左端补足；算术移位（arithmetic shift），空出的位置用符号位填充（见本节末的问答环节）。操作示例如下图所示。

| 值<br>典型常量 | 32位整数                               |        |     |        |    |        |        |
|-----------|-------------------------------------|--------|-----|--------|----|--------|--------|
|           | 0b000000000000000000000000000001111 | 0b1111 | 0xF | 0x1234 |    |        |        |
| 操作        | 取反                                  | 按位与    | 按位或 | 按位异或   | 左移 | 右移（逻辑） | 右移（算术） |
| 操作符       | ~                                   | &      |     | ^      | << | >>>    | >>     |

Java 内置 int 数据类型的位操作操作符

891

移位和掩码。这类操作的主要用途之一是掩码 (masking)，使用掩码可以将某一位或某一组位与同一字中的其他位区分开。通常情况下更愿意用十六进制常量表示掩码。例如，掩码 0x80000000 可以用于在 32 位字中隔离最左端一位，掩码 0x000000FF 可用于隔离最右边的 8 位，掩码 0x007FFFFFFF 可用于隔离最右端的 23 位。

再深入一点，我们通常使用移位和掩码提取一组连续位所表示的整数值，如下：

- 使用右移指令将位放置于最右端。
- 如果我们想要  $k$  位，创建一个常量掩码，最右端  $k$  位为 1，其余位皆为 0。
- 使用按位与操作，并将这  $k$  位与其他数据分开。掩码中的 0 会使结果中都为 0，掩码中的 1 会使结果中出现我们感兴趣的位。

这个操作序列使我们能够像使用任何其他 int 值一样使用结果，这通常也正是我们想要的。在本章的后面我们将关注移位和掩码以隔离十六进制数字，如下面右图所示。

取反

~ 01010001110101110000000000001111  
1010111000101000111111111110000

按位与

01010001110101110000000000001111  
& 00110001011011100011000101101110  
00010001010001100000000000001110

按位或

01010001110101110000000000001111  
| 00110001011011100011000101101110  
01110001111111110011000101101111

按位异或

01010001110101110000000000001111  
^ 00110001011011100011000101101110  
01100000101110010011000101100001

左移6位

01010001110101110000000000001111  
<<0000000000000000000000000000110  
01110101110000000000001111000000

右移6位

01010001110101110000000000001111  
>>0000000000000000000000000000011  
00001010001110101110000000000001

移位和按位操作 ( 32 位 )

| x | y | AND(x,y) | OR(x,y) | XOR(x,y) | 表达式                     | 值          |
|---|---|----------|---------|----------|-------------------------|------------|
| 0 | 0 | 0        | 0       | 0        | 0x00008A2B >> 8         | 0x0000008A |
| 0 | 1 | 0        | 1       | 1        | (0x00008A2B >> 8) & 0xF | 0x0000000A |
| 1 | 0 | 0        | 1       | 1        | 移位和掩码示例                 |            |
| 1 | 1 | 1        | 1       | 0        |                         |            |

892 按位操作定义的真值表

作为按位操作的实际应用示例，程序 6.1.2 说明了如何使用移位和掩码来从浮点数中提取符号、指数和小数。大多数计算机用户可以不必处理此级别的数据表示，仍然能够顺利地  
完成自己的编程任务（实际上在本书中，至今为止我们根本就不需要这样做），但是，如你所见，位操作在各种应用中起着重要的作用。

字符 为了处理文本，我们需要对字符进行二进制编码。基本方法相当简单：有一个表定义了字符和  $n$  位无符号二进制整数之间的对应关系。如果我们使用 6 位，可以编码 64 个不同的字符；使用 7 位，可以编码 128 个不同字符；使用 8 位，可以编码 256 个不同字符等。与浮点数一样，随着计算机的演变，许多不同的方案投入使用，人们在不同情况下仍然在使用各种不同的编码。

ASCII。美国信息交换标准代码 (American Standard Code for Information Interchange, ASCII) 在 20 世纪 60 年代被作为标准提出，从那以后便得到广泛使用。虽然在现代计算中通常用 8 位的字节存储数据，但它是一种 7 位码，最前一位通常会被忽略。

程序6.1.2 提取浮点数的组成部分

```
public class ExtractFloat
{
 public static void main(String[] args)
 {
 float x = Float.parseFloat(args[0]);
 int t = Float.floatToIntBits(x);

 if ((t & 80000000) == 1) StdOut.println("Sign: -");
 else StdOut.println("Sign: +");

 int exponent = ((t >> 23) & 0xFF) - 127;
 StdOut.println("Exponent: " + exp);

 double fraction = 1.0 * (t & 0x007FFFFF) / (1 << 23);
 double mantissa = 1.0 + fraction;
 StdOut.println("Mantissa: " + mantissa);

 StdOut.println(mantissa * (1 << exponent));
 }
}
```

该程序表明了Java位操作的用法，对从命令行参数中输入的浮点数提取其符号、指数和小数，然后使用指数和小数重新计算数字的绝对值。

```
% java ExtractFloat -100.25
Sign: -
Exponent: 6
Mantissa: 1.56640625
100.25
```

ASCII 码被提出的初衷是通过电传打字机（teletypewriter）进行通信，电传打字机可以发送并接收文本。因此，许多编码字符都是这种机器的控制字符。一些控制字符是通信协议的一部分（例如，ACK 意味着“确认”），其他字符控制机器的输出（例如，BS 意味着“退格”，CR 代表“回车”）。

下表是 ASCII 码的一种定义，其中提供了 8 位二进制码（也就是 2 位十六进制）与一个字符之间相互转换的对应关系。使用第一个十六进制数字索引行，第二个十六进制数字索引列，可以找到其编码的字符。例如，31 编码了数字 1，4A 编码了字母 J 等。该表为 7 位 ASCII 码，所以第一位十六进制数字必须小于或等于 7。以 0 或 1 开头的十六进制数字（以及 20 和 7F）对应的 ASCII 码是非打印控制字符，如 CR，现在意味着“换行符”（其他大部分已经很少在现代计算中使用）。

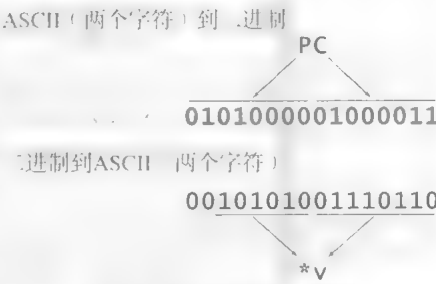
|    | _0  | _1  | _2  | _3  | _4  | _5  | _6  | _7  | _8  | _9 | _A  | _B  | _C | _D | _E | _F  |
|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|-----|-----|----|----|----|-----|
| 0_ | NUL | SOH | SIX | ETX | EOT | ENQ | ACK | BEL | BS  | HT | LF  | VT  | FF | CR | SO | SI  |
| 1_ | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US  |
| 2_ | SP  | !   | "   | #   | \$  | %   | &   | '   | (   | )  | *   | +   | ,  | -  | .  | /   |
| 3_ | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9  | :   | ;   | <  | =  | >  | ?   |
| 4_ | @   | A   | B   | C   | D   | E   | F   | G   | H   | I  | J   | K   | L  | M  | N  | O   |
| 5_ | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y  | Z   | [   | \  | ]  | ^  | _   |
| 6_ | `   | a   | b   | c   | d   | e   | f   | g   | h   | i  | j   | k   | l  | m  | n  | o   |
| 7_ | p   | q   | r   | s   | t   | u   | v   | w   | x   | y  | z   | {   |    | }  | ~  | DEL |

十六进制与 ASCII 码之间的转换表



Unicode。在 21 世纪的互联网世界中，常常需要处理比来自 20 世纪的 ASCII 字符更多的字符，所以一种新的名为 Unicode 的编码标准出现。通过对大多数字符使用 16 位（对于某些字符最多可以使用 24 位或 32 位），Unicode 可以支持数以万计的字符和大量的世界语言。UTF-8 编码（能够将字符序列编码为 8 位字节序列，反之亦然，大多数字符可映射到两字节）迅速作为一种标准出现。这些规则复杂而又全面，它们在大多数现代系统（如 Java）中得到了支持，所以程序员通常不用担心细节。ASCII 在 Unicode 内得到了兼容：所有 ASCII 字符映射到 1 字节，因此 ASCII 文件是 UTF-8 文件的特殊情况（向后兼容）。

我们通常将尽可能多的信息打包到一个计算机字中，因此可以用 16 位编码两个 ASCII 字符（如右图所示）、32 位编码（四个字符）、64 位编码（八个字符）等。在 Java 这样的高级语言中，这些细节和 UTF-8 编解码在 String 数据类型中实现，在本书中会一直使用。然而，对于 Java 程序员而言，更重要的是了解一些关于信息表示的基本事实，因为它会影响程序的资源需求。例如，许多程序员发现，自 2000 年当 Java 从 ASCII 切换到 Unicode 时，他们程序的内存使用量突然翻倍，并开始使用 16 位的 char 来对每个 ASCII 字符进行编码。有经验的程序员知道如何在每个 char 中打包两个 ASCII 字符，以便在必要时节省内存。



ASCII 与二进制转换示例

895

**总结** 一般来说，编写能够正确独立于数据表示的程序是明智之举。许多编程语言完全支持这一观点。不幸的是，通过这种方法得出的硬件使用方式，则直接站在了充分利用计算机能力的对立面。Java 的原始类型旨在支持这一观点。例如，如果计算机拥有能够实现 64 位整数加法或乘法的硬件，那么我们希望将每个加法或乘法运算减少到单个指令，以便我们的程序尽可能快地运行。因此，程序员尝试将具有性能关键操作的数据类型与在计算机硬件中实现的原始类型进行匹配是明智的。实现实际匹配可能会需要更深入地了解你的系统及其软件，但为了获得最佳性能这肯定是值得的。

你编写的程序可能一直是在使用各种数据类型进行计算。我们在本节中提供的消息是，由于每个比特序列都可以以许多不同的方式解释，计算机内任何给定的比特序列的含义取决于上下文。你可以编写程序并以任何你想要的方式来解释位。随着编程的过程你会看到，你不能只根据使用的位数来判断它们的数据类型，甚至都不能确定它们是不是数据。

为了进一步强调这一点，下表给出了几个不同的 16 位值，以及如果被解释为二进制整数、十六进制整数、无符号整数、二进制补码整数、binary16 浮点数和 ASCII 字符对应的值。这些只是在计算机中表示信息的无数可用方法的几个较为简单的例子。

| 二进制              | 十六进制 | 无符号整数  | 二进制补码   | 浮点数 (binary16)               | ASCII 字符 |
|------------------|------|--------|---------|------------------------------|----------|
| 0001001000110100 | 1234 | 4 660  | 4 660   | 0.0007572174072265625        | DC2 4    |
| 1111111111111111 | FFFF | 65 535 | -1      | -131008.0                    | DEL DEL  |
| 1111101011001110 | FACE | 64 206 | -1 330  | -55744.0                     | e N      |
| 0101011001000100 | 5644 | 22 052 | 22 052  | 100.25                       | V D      |
| 1000000000000001 | 8001 | 32 769 | -32 767 | -0.0000305473804473876953125 | NUL SOH  |
| 0101000001000011 | 5043 | 20 547 | 20 547  | 34.09375                     | P C      |
| 0001110010101011 | 1CAB | 7 339  | 7 339   | 0.004558563232421875         | FS +     |

解释 16 位值的 6 种不同方式

896

## 问答环节

问：如何找出计算机中字的大小？

答：需要找到处理器名称，然后查找处理器规格。你最可能拥有的的是一个 64 位的处理器。如果不是，是时候换一台新计算机了。

问：在大多数计算机都是用 64 位字的情况下，为什么 Java 的 int 值使用 32 位？

答：这是很久以前的设计决策。Java 不同于其他常规的编程语言，因为它完全指定了 int 的表示形式。这样做的优点是，对于旧的程序，比起那些依赖硬件细节设定 int 的表示方式的语言，Java 程序在新的计算机上工作的可能性更高。缺点是 32 位通常还不够。例如，2014 年，Google 不得不改变其记录视频观看次数的 32 位变量，因为歌曲视频《江南 style》(Gangnam Style) 被观看超过 2 147 483 647 次。在 Java 中，你可以切换到 long。

问：这似乎应该是由系统处理的事情，对吧？

答：有些语言，如 Python，对于整数的大小不做限制，当需要时，系统可以使用多个字表示整数值。在 Java 中，你可以使用 BigInteger 类。

问：什么是 BigInteger 类？

答：它允许你使用整数进行计算而不用担心溢出。例如，如果你导入 java.math.BigInteger，那么代码

```
BigInteger x = new BigInteger("2");
StdOut.println(x.pow(100));
```

输出 1267650600228229401496703205376，即  $2^{100}$ 。你可以将 BigInteger 视为一个字符串（内部表示比这个更有效率）。该类提供了标准算术操作及其他操作的方法。例如，该方法在加密时很有用，因为在一些加解密系统中的关键操作常常需要数百位数字的算术运算。在库函数的实现中，必要时会使用多个数字进行连接，所以溢出不是一个问题。当然，其操作比内置的 long 或 int 操作要复杂得多，所以 Java 程序员不会对 long 或 int 就可以满足支持的整数使用 BigInteger。

[897]

问：为什么是十六进制？其他进制都不能完成这项工作吗？

答：基数为 8，或八进制，在早期 12 位、24 位或 36 位的计算机中广泛使用，因为一个字的内容可以用 4、8 或 12 位八进制数字表示。在这样的系统中使用八进制的优点是只需要使用十进制数字 0~7，所以简单的 I/O 设备（如数字键盘）十进制和八进制都可以使用。但是八进制对于 32 位和 64 位的字来说并不方便，因为字的大小不能被 3 整除（也不能被 5 或 6 整除，所以换用更大的基数也不可能）。

问：如何防范溢出？

答：这并不简单，因为每种操作都需要不同的检查。例如，如果你知道 x 和 y 都为正，想要计算  $x + y$ ，可以检查  $x < \text{Integer.MAX\_VALUE} - y$ 。

答：另一种方法是“向上”转换为一种范围更大的类型。例如，如果你正在用 int 类型计算，则可以将其转换为 long 类型，然后将结果转换回 int（如果结果不是很大的话）。

答：在 Java 8 中，可以使用 Math.addExact()。对于溢出的情况，这个操作可以抛出异常。

问：对于二进制补码，硬件如何检测溢出？

答：规则很简单，虽然证明有点棘手：检查最左位的进位进项及前导位的进位出项。如果它们不同，则显示溢出（参见下图）。

进位进项  
与进位出  
项不同  
^

```

100000000000000000
1111111111111000 -8
10000000000000100 -32764
1111111111111100 -4 X

```

进位进项  
与进位出  
项不同  
^

```

0111111111111000
0111111111111000 32760
0000000000001000 + 8
1000000000000000 -32768 X

```

溢出 (16 位二进制补码)

问：算术移位的目的是什么？

答：对于二进制补码整数，算术移位右移 1 位与整数除 2 的结果相同。例如，如右图所示， $(-16) \gg 3$ ，结果为 -2。为了检验你对该操作的理解，计算  $(-3) \gg 1$  和  $(-1) \gg 1$  的值。这种转换被称为“符号扩展”(sign extension)，与逻辑移位的“零扩展”相反。

问：如果 y 为负或 y 大于 31，则  $x \gg y$  的结果是什么？

答：对于移位操作，Java 只使用第二个操作数的低五位。这种行为与典型计算机上的物理硬件吻合。

正数

```

x: 00000000000010000 16
x >> 3: 0000000000000010 2
 ↑
 用0补位

```

负数

```

x: 11111111111110000 -16
x >> 3: 1111111111111110 -2
 ↑
 用1补位

```

算术移位 (16 位二进制补码)

问：我没有真正理解 1.2 节中问答环节的例子，为什么  $(0.1+0.1==0.2)$  为真，而  $(0.1+0.1+0.1===0.3)$  为假。可以详细说明一下吗？

答：在 Java 源码中，像 0.1 或 0.3 的常量会被转换为最接近的 64 位 IEEE 754 数字（当其偶数时，最低有效位为 0），是一个 Java 的 double 值。下面是 0.1、0.2 和 0.3 的常量：

| 常量  | 最接近的64位IEEE 754数字                                          |
|-----|------------------------------------------------------------|
| 0.1 | 0.1000000000000000055511151231257827021181583404541015625  |
| 0.2 | 0.20000000000000000111022302462515654042363166809082031250 |
| 0.3 | 0.29999999999999999888977697537484345957636833190917968750 |

如你所见， $0.1 + 0.1$  等于 0.2，但  $0.1 + 0.1 + 0.1$  比 0.3 大。这种情况与下面的情形类似，

[899] 对于整数， $2/5 + 2/5$  等于  $4/5$ （它们都是 0），但是  $2/5 + 2/5 + 2/5$  不等于  $6/5$ 。

问：System.out.println(0.1) 输出 0.1，而非前文表格中的值。为什么？

答：很少有程序员需要这么高的精度，所以 println() 出于可读性考虑将数值截短。其实现方式是将打印的精度控制到能与 double 类型的相邻值区分开来，然后在这个精度的基础上再多显示一位数字。你可以使用 printf() 更精确地控制显示格式，而 BigDecimal 则可用于进一步扩展精度。例如，你可以导入 java.math.BigDecimal，那么代码

```
double x = 0.1;
StdOut.println(new BigDecimal(x));
```

[900] 会输出 0.1000000000000000055511151231257827021181583404541015625。

练习

- 6.1.1 将十进制数字 92 转换为二进制。  
答案：1011100。
- 6.1.2 将八进制数字 31415 转换为二进制。  
答案：011001100001101。
- 6.1.3 将八进制数字 314159 转换为十进制。  
答案：这不是一个八进制数字！你可以进行计算，甚至是使用 Convert，得到结果 104561，但是 9 不是一个合法的八进制数字。在本书网站上的 Convert 版本包含这样的合法习惯检查。老师在考试中考查这一点并不罕见，所以要小心！
- 6.1.4 将十六进制数字 BB23A 转换为八进制。  
答案：首先转换为二进制 1011 1011 0010 0011 1010，然后一次考虑 3 位：10 111 011 001 000 111 010，可以将其转换为八进制 2731072。
- 6.1.5 将两个十六进制数字 23AC 和 4B80 相加，并将结果用十六进制表示。提示：直接使用十六进制相加，而非转换为十进制，相加，再转换回来。
- 6.1.6 假设  $m$  和  $n$  均为正整数。在  $2^{m+n}$  的二进制表示中有多少位为 1？
- 6.1.7 哪个十进制整数的十六进制表示正好是它本身反过来？  
答案：53 的十六进制是 35。
- 6.1.8 证明：一次将十六进制数字的一位转换为二进制，得到的总是正确结果，反之亦然。
- 6.1.9 IPv4 是 20 世纪 70 年代提出的协议，其规定了互联网上的计算机如何进行通信。互联网上的每台计算机都需要自己的 Internet 地址。IPv4 使用 32 位地址。互联网可以容纳多少台计算机？是否足够让每个手机和每台烤面包机都有自己的地址？
- 6.1.10 在 IPv6 协议中每台计算机都有一个 128 位的地址。如果该标准被采用，互联网上可以容纳多少台计算机？是否足够？  
答案： $2^{128}$ 。至少短期内足够——地球表面每平方微米可以分到 5000 个地址。
- 6.1.11 在表中填写各表示所代表的值：

| 表示 | $\sim 0xFF$ | $0x3 \& 0x5$ | $0x3   0x5$ | $0x3 \wedge 0x5$ | $0x1234 \ll 8$ |
|----|-------------|--------------|-------------|------------------|----------------|
| 值  |             |              |             |                  |                |

- 6.1.12 实现正文中指定的 toInt() 方法，用于将 0~9 或 A~Z 范围内的字符转换为 0~35 之间的值。  
答案：  

```
public static int toInt(char c)
{
 if ((c >= '0') && (c <= '9')) return c - '0';
 return c - 'A' + 10;
}
```
- 6.1.13 实现正文中指定的 toChar() 方法，用于将 0~35 之间的值转换为 0~9 或 A~Z 范围内的字符。  
答案：

```
public static char toChar(int i)
{
 if (i < 10) return (char) ('0' + i);
 return (char) ('A' + i - 10);
}
```

6.1.14 修改 Convert (以及前两个练习的答案), 使其可以使用 long 型, 检测溢出并检查输入字符串中的数字是否在基数指定的范围内。

答案: 见本书网站上的 Convert.java。

6.1.15 将某版本的 toString() 方法添加到 Convert 中, 该 toString() 方法的第三个参数指定了生成的字符串的长度。如果指定的长度小于转换出的位数, 则只返回最右端的数字; 如果指定长度更大, 则前端用 0 补足。例如, toString(64206,16,3) 应返回 “ACE”, toString(15,16,4) 应返回 “000F”。提示: 先调用两个参数的版本。

6.1.16 构建一个 Java 程序 TwosComplement, 从命令行中获取一个 int 型值 i 和字长 w, 输出 i 的 w 位二进制补码表示及这个数字的十六进制表示。假设 w 是 4 的倍数。例如, 你的程序应该有如下表现:

```
% java TwosComplement -1 16
1111111111111111 FFFF
% java TwosComplement 45 8
00101101 2D
% java TwosComplement -1024 32
111111111111111111110000000000 FFFFC00
```

6.1.17 将 ExtractFloat 修改为程序 ExtractDouble, 使其能够对 double 类型值实现相同的操作。

6.1.18 编写一个 Java 程序 EncodeDouble, 从命令行获取一个 double 类型值, 根据 IEEE 754 binary32 标准编码为一个浮点数。

903

6.1.19 请填写表中的空白处。

| 二进制              | 浮点数     |
|------------------|---------|
| 0010001000110100 |         |
| 1000000000000000 |         |
|                  | 7.09375 |
|                  | 1024    |

6.1.20 请填写表中的空白处。

| 二进制              | 十六进制 | 无符号数 | 二进制补码 | ASCII 字符 |
|------------------|------|------|-------|----------|
| 1001000110100111 |      |      |       |          |
|                  | 9201 |      |       |          |
|                  |      | 1000 |       |          |
|                  |      |      | - 131 |          |
|                  |      |      |       | ??       |

904

创新练习

6.1.21 IP 地址和 IP 值。一个 IP 地址 (IPv4) 由整数 w、x、y、z 组成, 通常被写成字符串 w.x.y.z。对应的 IP 值为  $16777216w + 65536x + 256y + z$ 。给定一个 IP 数字 n, 通过  $w = (n/1677216) \bmod 256$ 、 $x = (n/65536) \bmod 256$ 、 $y = (n/256) \bmod 256$ 、 $z = n \bmod 256$  可以得到其对应的 IP 地址。编写一个函数, 输入一个 IP 值, 返回表示 IP 地址的字符串, 编写另一个函数, 输入 IP 地址, 返回对应 IP 值的 int 值。例如, 给定 3401190660, 第一个函数应该返回 202.186.13.4。

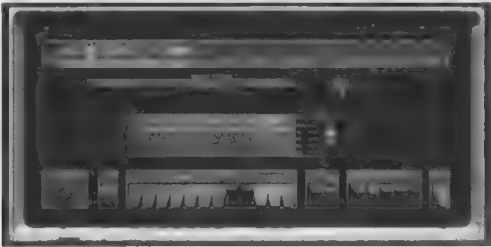
- 6.1.22 IP 地址。编写一个程序，从命令行中获取一个 32 位字符串，并以点分十进制的形式（dotted decimal form）输出其对应的 IP 地址。也就是说，一次获取 8 位，将每一组转换为十进制，并将每组以点分隔开。例如，二进制 IP 地址 01010000000100000000000000000001 应被转换为 80.16.0.1。
- 6.1.23 MAC 地址。编写一个函数，实现 MAC 地址与 48 位 long 型整数的相互转换。
- 6.1.24 Base64 编码。Base64 编码是通过互联网发送二进制数据的常用方法。它将任意数据转换为 ASCII 文本，使其可以在系统之间通过电子邮件发送，而不会出现任何问题。编写一个程序来读取一个任意的二进制文件，并使用 Base64 进行编码。
- 6.1.25 浮点数软件。编写一个 FloatingPoint 类，它共有三个实例变量 sign、exponent 和 fraction。实现加法和乘法，以及 toString() 和 parseFloat()，支持 16 位、32 位和 64 位格式。
- 6.1.26 DNA 编码。编写一个 DNA 类，支持仅由 A、T、C 或 G 字符组成的字符串的高效表示。其包含一个将字符串转换为内部表示形式的构造函数、将内部表示形式转换为字符串的 toString() 方法、返回指定索引处字符的 charAt() 方法，以及将 String pattern 作为参数并返回 pattern 在表示字符串中第一次出现的 indexOf() 方法。对于内部表示，请使用 int 型数组，每个 int 中包含 16 个字符（每个字符占两位）。

905

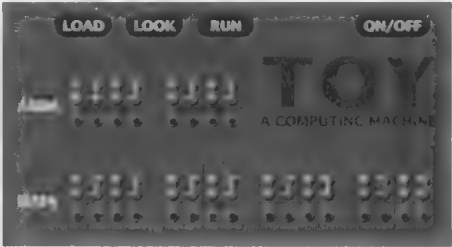
6.2 TOY 计算机

为了帮助你更好地理解计算机上计算的本质，我们在本节中介绍 TOY，这是一个为本书设计的假想计算机，与 20 世纪 70 年代首次广泛使用的计算机非常相似。我们今天研究它，因为它也具有现代微处理器所具有的基本特征，如移动设备的处理器、计算机中的处理器以及其他任何地方出现的处理器，甚至包括在这些年间开发的无数其他计算设备。下图展示了 PDP-8——一台 20 世纪 70 年代的真实计算机，以及我们的假想计算机 TOY。

PDP-8, 20世纪70年代



TOY, 假想计算机



真实的计算机与假想机

TOY 演示了简单计算模型如何执行那些关键的计算任务，也可以帮助你了解计算机的基本特征。在过去的几十年中，计算演变的一个显著事实是，所有的计算机都具有相同的基本结构，这种方法在约翰·冯·诺依曼 1945 年首次阐述之后几乎立即被广泛采用。

我们首先介绍 TOY 计算机的基本组成部分。TOY 其实仅有几个部分，且每部分的目的都很容易理解。所有的计算机都由相似的部分构成。

接下来我们描述 TOY 计算机的使用和编程方法。我们从上一节中介绍的信息表示的基本方法开始，先来看对这类信息的操作。换句话说，我们正在使用 TOY 计算机硬件实现对数据类型（即值的集合以及对这些值的操作）的支持。在这个层面上的工作被称为机器语言编程。学习在机器语言层面上编程会帮助你更好地理解 Java 程序与计算机之间的关系，以及计算本身的性质。机器语言编程实际上在诸如视频处理、音频处理和科学计算等性能关键

906

类的应用中仍然使用。你将会看到，学习如何用机器语言编程并不困难。

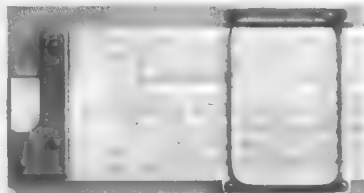
第7章将介绍如何用硬件构建这样的机器。这是揭开计算机神秘面纱的最后一步，能够帮助你更好地理解 Java 程序与物理世界之间的联系。

在所有计算机中都能找到一个重要的抽象层，即机器语言。它能够对处理器执行的操作进行准确描述，像 Java 这样的高级语言编写的程序最终会被翻译成这样的语言来执行，同时，它也提供了构建和实现该机器的电路蓝图。

**简要的历史回顾** 想象一个没有计算机的现代世界可能有点儿困难。我们选择一个时间点，如 20 世纪 50 年代，当时经历二战后的世界正逐步走向工业化，并研制出了汽车、飞机、卫星等各种现在仍广泛使用的技术，但普通人甚至一般的科学家或工程师都没有计算机可以利用。

构建计算机的最初动机是为了执行科学、工程和商业等各种领域中的计算。“二战”本身就证明了这一点，从约翰·冯·诺依曼计算的弹道表到艾伦·图灵开发的密码机（恩尼格玛密码机——译者注），更不用说在洛斯阿拉莫斯（Los Alamos）完成的推动原子弹发展的计算。想象一下，在没有计算机的情况下如何经营一家银行或制造一辆汽车。

在 20 世纪 70 年代之前一个典型的科学或工程学生用于计算的最重要的工具是算尺（slide rule，见右图），这是一种绝对非电子、非数字化的设备，但非常有用，特别是对于计算对数和乘法运算。另一个常用工具是一本《函数表》（tables of functions），例如，要计算  $\sin(x)$ ，就要在书中查找！



算尺

起初人们开始使用计算机时，通常是由一组人一起使用，并且使用起来非常麻烦。尽管如此，相对于算尺和函数表来说计算机是一个巨大的改进。在很短的时间内，人们共享大型计算机的方式使得算尺和函数表成为过去式。多年来，人们也使用与计算机拥有相同技术的计算器，但计算器只是用于计算，并且是手持式的。当然，对于简单的计算，人们使用计算器仍然能够实现。

就像现在一样，对于复杂的计算，科学家、工程师和应用程序开发人员通过编写计算机程序来解决问题。以此为目的创造出的第一批设备的基本设计至今仍然改变着世界，这是非常了不起的。

**TOY 计算机的组件** 我们首先对半个世纪以来几乎在所有计算机中使用的基本设计组件进行一个概述，但是只简要说明我们的 TOY 计算机涉及的必要零件。

**内存。**对任何计算机来说，内存（memory）都是重要的组成部分。它不仅存储着要处理的数据和计算结果，还存储着机器运行的程序。TOY 的内存由 256 个字组成，每个 16 位。按照今天的标准，这当然不算什么，但你会对它能支持的计算范围感到惊讶。这是一个发人深省的思考： $256 \times 16 = 4096$  位，有  $2^{4096}$  个不同的可能值，所以 TOY 可以做的绝大多数事情将永远不会在这个世界上发生。

若用十六进制表示法，我们可以用 4 个十六进制数字来指定一个内存字的内容。此外，我们考虑将字从 0 到 255 进行编号，以便可以将每个字用两位十六进制数表示，这个编号称为地址。例如，我们可以说“地址 1E 上存储的字的值是 0FA2”或者“存储单元 1E 的值是 0FA2”。为了表述方便，我们经常使用数组符号，也就是“M[1E] 是 0FA2”。我们可以使用像上述那样的 16 行表来指定 TOY 内存的内容。第一列给出位置 00 到 0F 的值；第二列给出位置 10 到 1F 的值，等等。这样的表被称为内存导出（memory dump）。与此同时，内存还必须能够存储我们在本章中设计的所有程序！



|    | 0_   | 1_   | 2_   | 3_   | 4_   | 5_   | 6_   | 7_   | 8_   | 9_   | A_   | B_   | C_   | D_   | E_   | F_   |
|----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| _0 | 7A10 | 8A15 | 8A2B | 7101 | 7101 | 7800 | 8AFF | 7101 | 7101 | BB0E | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _1 | 7BEF | 8B16 | 8B2C | 75FF | 7A00 | 8CFF | 8BFF | A90A | A90A | 1EE1 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _2 | 9AFF | 1CAB | 2CAB | 7901 | 7B01 | CC55 | 7101 | 140A | 180A | 900E | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _3 | 9BFF | 9C17 | CC29 | 2C59 | 894C | 188C | 7900 | 7B00 | C98F | 1EE1 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _4 | 7101 | 0000 | DC27 | CC3B | C94B | C051 | 22B9 | C97C | AC09 | 900E | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _5 | 7900 | 0008 | 2BBA | 1991 | 9AFF | 98FF | C26B | 2991 | 2CBC | 1EE1 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _6 | 22B9 | 0005 | C022 | 1A09 | 1CAB | 0000 | 1CA9 | 2441 | CC96 | EF00 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _7 | C200 | 0000 | EF00 | 8B3D | 1AB0 | 0000 | 8DFF | AC04 | 1991 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _8 | 1CA9 | 0000 | 0000 | FF22 | 1BC0 | 0000 | BD0C | 2EBC | 1809 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _9 | AD0C | 0000 | 00C3 | 2AA1 | 2991 | 0000 | 1991 | DE7B | A909 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _A | 9DFF | 0000 | 0111 | CA36 | C044 | 0000 | C064 | 1B0C | DCBE | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _B | 1441 | 0000 | 0000 | 9B3E | 0000 | 0000 | 1AA9 | C074 | 1981 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _C | C006 | 0000 | 0000 | 0000 | 0000 | 000C | FF60 | BB0A | EF00 | 1809 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _D | 0000 | 0000 | 0000 | 005B | 0000 | FF70 | EF00 | 0000 | A909 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _E | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 9BFF | 0000 | 0000 | C083 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| _F | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | BE08 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |

TOY 计算机的内存导出内容 (256 个 16 位的字)

指令。TOY 计算机 (以及几乎所有其他计算机) 的一个关键特征是, 内存字的内容可以解释为数据, 也可以解释为指令, 这取决于上下文。例如, 从上一节知道, 值 1234 可能被解释为表示整数  $4660_{10}$  或实数  $0.00302886962890625$ 。在本节中, 你将会了解到, 它也可能代表将两个数字相加的机器指令。程序员要确保数据被视为数据, 并将指令视为指令。我们将分析所有的 TOY 指令是如何编码的, 以便知道如何将任何一个 16 位值解码为指令 (以及如何将任何一个指令编码为 16 位值)。

寄存器。寄存器是保存一系列位的机器组件, 更像是内存中的字。寄存器用于在计算过程中保存中间结果。你可以认为它们在 TOY 编程中扮演变量的角色。TOY 有 16 个寄存器, 从 0 到 F 编号。与存储器一样我们使用数组符号为其命名, 即从 R[0] 到 R[F] 的寄存器。由于它们是 16 位的, 与内存字相同, 所以我们用 4 位十六进制值表示每个寄存器的内容, 并用 16 个 4 位十六进制数来表示所有寄存器的内容。右边的表格展示了典型计算过程中寄存器的内容, 我们将在后面讨论。按照惯例, R[0] 总是 0000。

|      |      |
|------|------|
| R[0] | 0000 |
| R[1] | 0001 |
| R[2] | 000A |
| R[3] | 0000 |
| R[4] | 0000 |
| R[5] | 0000 |
| R[6] | 0030 |
| R[7] | 0000 |
| R[8] | 003A |
| R[9] | 0000 |
| R[A] | 0A23 |
| R[B] | 0B44 |
| R[C] | 0C78 |
| R[D] | 0000 |
| R[E] | 0000 |
| R[F] | 0000 |

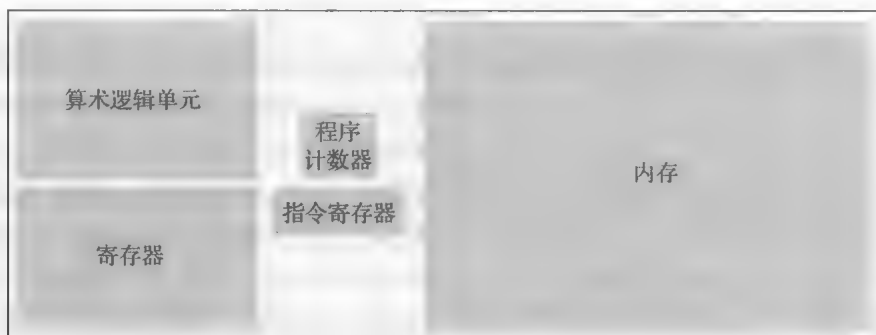
908  
909

算术逻辑单元。算术逻辑单元 (Arithmetic Logic Unit, ALU) 是 TOY 的计算引擎——机器执行所有计算的主力。通常, 一条 TOY 指令指示 ALU 计算某个函数, 它将两个寄存器作为参数, 并将结果存入第三个寄存器。例如, TOY 指令 1234 表示将 R[2] 和 R[3] 的内容送入 ALU, 将它们相加, 然后将结果写入 R[4]。在本节的后面, 我们将介绍如何根据这些指令来编写 TOY 程序。

TOY 的寄存器

程序计数器和指令寄存器。程序计数器 (Program Counter, PC) 是一个 8 位的内部寄存器, 用于保存下一条要执行的指令的地址。指令寄存器 (Instruction Register, IR) 是一个保存正在执行的当前指令的 16 位内部寄存器。这些寄存器是机器操作的核心。虽然 IR 不直接被程序访问, 但是程序员总能知道它的内容。

下图包含了这些基本的组件。在第 7 章中, 我们将考虑如何创建一个电路 (circuit) 以实现它们的功能。在本章的其他部分, 我们将探讨它们是如何操作的, 以及程序员如何控制它们以执行期望的计算。



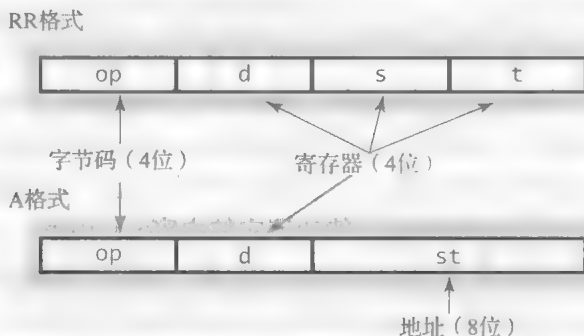
TOY 计算机的组件

**读取 - 递增 - 执行周期** TOY 计算机通过重复执行特定的动作序列来执行指令。首先检查 PC 的值，并将该存储位置的内容提取（复制）到 IR 中。接下来，它将程序计数器递增 1（例如，如果程序计数器当前为 10，则递增到 11）。最后，它将 IR 中的 16 位值解释为指令，并按照 TOY 计算机的规则执行（该规则我们稍后描述）。每条指令都可以修改各种寄存器、主存储器甚至程序计数器本身的内容。执行该指令后，机器重复整个读取 - 递增 - 执行周期，使用新的程序计数器的值找到下一条指令。这个过程一直持续下去，或者直到机器执行停机指令。与 Java 一样，我们可以编写出无限循环的程序。为了使 TOY 计算机在无限循环中停止，程序员必须将其关闭，甚至拔掉它的电源。



**指令** 任何 16 位值（任何内存字的内容）都可以解释为一条 TOY 指令，每条指令的目的是以某种方式修改机器的状态（内存字、寄存器或 PC 的值）。为了描述指令的操作，我们使用伪代码（pseudo-code），除了直接操作内存字、寄存器和 PC 以外，它与 Java 代码非常类似。

**一条指令的剖析。** 我们使用十六进制编码表示指令：每个 16 位指令是 4 个十六进制数字。指令的第一位是它的操作码（opcode），指定所执行的操作。共有 16 个不同的指令，每一个都由一个十六进制数字表示。指令的第二位数字指定一个寄存器（register）——每条指令都使用或更改某个寄存器的值。共有 16 个寄存器，同样地每一个都由一个十六进制数字表示。大部分指令的第三位和第四位十六进制数字都以以下两种指令格式（instruction format）编码：在 RR 格式的指令中，剩下的两个十六进制数字中的每一个都是指一个寄存器。在 A 格式的指令中，第三个和第四个十六进制数字（一起）指定一个内存地址。



TOY 指令剖析

指令集。下面的表格描述了 TOY 的所有指令。这个表是 TOY 编程的完整参考指南，在编写 TOY 程序时请参考。接下来，我们将详细介绍这些指令。

911

| 操作码 | 描述          | 格式 | 伪代码                                             |
|-----|-------------|----|-------------------------------------------------|
| 0   | 停机          | -  |                                                 |
| 1   | 加           | RR | $R[d] \leftarrow R[s] + R[t]$                   |
| 2   | 减           | RR | $R[d] \leftarrow R[s] - R[t]$                   |
| 3   | 按位与         | RR | $R[d] \leftarrow R[s] \& R[t]$                  |
| 4   | 按位异或        | RR | $R[d] \leftarrow R[s] \wedge R[t]$              |
| 5   | 左移          | RR | $R[d] \leftarrow R[s] \ll R[t]$                 |
| 6   | 右移          | RR | $R[d] \leftarrow R[s] \gg R[t]$                 |
| 7   | 加载地址        | A  | $R[d] \leftarrow \text{addr}$                   |
| 8   | 加载          | A  | $R[d] \leftarrow M[\text{addr}]$                |
| 9   | 存储          | A  | $M[\text{addr}] \leftarrow R[d]$                |
| A   | 间接加载        | RR | $R[d] \leftarrow M[R[t]]$                       |
| B   | 间接存储        | RR | $M[R[t]] \leftarrow R[d]$                       |
| C   | 等于零则跳转      | A  | if ( $R[d] == 0$ ) $PC \leftarrow \text{addr}$  |
| D   | 正数则跳转       | A  | if ( $R[d] > 0$ ) $PC \leftarrow \text{addr}$   |
| E   | 跳转到寄存器指令的地址 | -  | $PC \leftarrow R[d]$                            |
| F   | 链接并跳转       | A  | $R[d] \leftarrow PC; PC \leftarrow \text{addr}$ |

TOY 指令集

停机。操作码 0 是最基本的指令，它简单地指示机器停机，即停止读取 - 递增 - 执行周期。此时，程序员可以检查内存的内容来查看计算结果。TOY 忽略停止操作指令的另外 3 个十六进制数字，所以 0000、0123 和 0FFF 都是暂停指令。

算术指令。操作码 1 和 2 是算术 (arithmetic) 指令，它调用 ALU 对两个寄存器 ( $R[s]$  和  $R[t]$ ) 进行算术运算，并将结果存入第三个寄存器 ( $R[d]$ )。例如，指令 1234 的意思是“将  $R[3]$  加到  $R[4]$  并把结果放在  $R[2]$  中”，2AAC 意思是“从  $R[A]$  中减去  $R[C]$  并把结果放在  $R[A]$  中”。这些指令可以说是实现了 TOY 的整型数据类型：值的集合是 16 位整数，操作是加和减。

内存地址指令。操作码 7、A 和 B 是内存地址 (memory address) 指令，我们用它们来操纵 TOY 中的地址。例如，指令 7423 的意思是“将  $R[4]$  设置为值 0023”或者  $R[4] = 0023$  (注意前导 0)。然后操作码 A 和 B 可以通过寄存器中的地址间接访问存储器。理解这些指令能够让你更有效地理解 Java 中变量访问的原理。我们在稍后讨论实现数组和链接结构时将更详细地讨论它们的用法。

操作码 7 的另一个重要用途是作为一个整型数据类型的指令：一旦这些位被加载到寄存器中，我们可以在算术指令中使用它们 (将它们当作一个整数来处理)。例如，我们使用指令 7C01 将  $R[C]$  设置为 0001 ( $R[C] = 0001$ )。

逻辑指令。操作码 3~6 是逻辑 (logical) 指令，它们调用 ALU 对寄存器中的位执行操作，就像我们在 5.1 节中提到的 Java 操作一样。操作码 3 是“按位与”，如果  $R[s]$  和  $R[t]$  中的相应位都是 1，则  $R[d]$  中的每位被设置为 1；否则，它被设置为 0。类似地，操作码 4 是“按位异或”，如果  $R[s]$  和  $R[t]$  中的相应位不同，则  $R[d]$  中的每位被设置为 1；操作码 5 是将  $R[s]$  中的位向左移  $R[t]$  位，并将结果保存在  $R[d]$  中，丢弃移出的位并按需要移入 0 位。操作码 6 与之相似，但是向右移位，并且移入的位与符号位匹配 (参见 6.1 节中的问答环节)。我们参照 Java 中对整数类型的设计，在逻辑指令中以相应的方式完成 TOY 的整型数据类型的实现。在 TOY 中，与 Java 一样，我们有时会忽略数据抽象规则，将数据视为 16 位序列，而不是当作一个整数。就像在 Java 代码中一样，移位和按位逻辑指令在实现和解码所有类型的数据时都非常有用。

912

内存指令。操作码 8 和 9 是内存 (memory) 指令, 用于在内存和寄存器之间传输数据。例如, 指令 8234 意思是“将内存字 M[34] 加载到寄存器 R[2] 中”, 或  $R[2] = M[34]$ 。而指令 9234 意味着“将 R[2] 的值存储到内存字 M[34] 中”或  $M[34] = R[2]$ 。

控制流指令。操作码 C~F 是控制流指令 (flow of control instruction), 控制流指令可以修改 PC, 对于实现流控制结构 (编程中的基本操作如条件、循环和函数等) 来说是必不可少的。例如, 指令 C212 的意思是“如果 R[2] 为 0, 则将 PC 设置为 12”, D212 的意思是“如果 R[2] 为正, 则将 PC 设置为 12”。特别要注意的是, 由于 R[0] 始终为零, 因此 C0xx 的意思是“将 PC 设置为 xx”。这个操作被称为无条件分支 (unconditional branch)。改变 PC 的值会达到改变控制流的效果, 因为下一条指令总是从 PC 给出的内存地址中获取。稍后当我们研究如何实现函数时, 将会更详细地分析操作码 E 和 F。

你对这套指令集的第一反应可能是它很小, 指令数目很少。这当然是对的, 但是本章的目标之一就是要说服你, 像这样的一小组指令可以编写相当于你在本书前部分学到的所有 Java 程序甚至所有程序。目前, 要记住的最重要的事情就是, 现在我们已经知道了如何将任何 16 位值解码为 TOY 指令。

```
按位与
0101000111010111
& 0011000101101110
0001000101000110

按位异或
0101000111010111
^ 0011000101101110
0110000010111001

向左位移6
0001000111010111
<< 0000000000000110
0111010111000000

向右位移3 (算术移位)
0000000111010111
>> 0000000000000011
0001000000111010

1000000111010111
>> 0000000000000011
1111000000111010

逻辑和位移指令
```

[913] 我们已经知道了如何将任何 16 位值解码为 TOY 指令。

**你的第一个 TOY 程序** 程序 6.2.1 是我们写的第一个 TOY 程序, 它实现了两个整数相加, 它就是你在 TOY 上的“Hello, World”。与 HelloWorld.java 一样, 在 1.1 节中, 我们从一个简单的程序开始, 让我们专注于运行程序的细节。程序 6.2.1 中的代码 (称为机器代码) 展示了我们用于 TOY 程序的各种约定:

- 包含与给定程序相关的所有数据和代码。
- 每行给出一个 2 位 (十六进制) 内存地址和该地址的 4 位 (十六进制) 值。
- PC 的起始值始终是第一条指令的地址, 用粗体突出显示。
- 第三列给出每个指令的伪代码。

程序本身就是存储在存储单元 10~14 中的 5 个 4 位十六进制数字。伪代码使得这个程序很容易理解——它读起来更像 Java 程序。

为了跟踪一个 TOY 程序, 我们简单地写下执行的每个指令的 PC 和 IR 值, 以及任何受影响的寄存器或执行后的内存字。代码下面的表格为程序 6.2.1 提供了这样一个跟踪。为了找到计算结果, 我们列出了到达停机指令时的内存内容、停机指令本身, 任何有变化的内存位置的值已经以粗体突出显示。在这个例子中, 只有一个存储器值发生变化: 位置 17 中存储着计算结果 000D。

这个过程看起来非常简单, 但是我们还没有描述真正让程序运行的过程。对于 Java, 我们可以描述如何使用编辑器创建程序的文件, 然后使用编译器和 Java 虚拟机执行它, 并在终端窗口中查看结果。对于 TOY 来说, 你不得不考虑没有操作系统、没有应用程序, 当然也没有编辑器、终端仿真器、编译器, 甚而运行时都没有键盘或者显示器的情况。

实际上, 程序 6.2.1 底部的图片展示了运行程序 6.2.1 的“结果”: 计算机前面板底部的指示灯显示计算结果 000D, 二进制中表示为 0000000000001101。接下来, 我们逐步分析程序在 TOY 计算机上运行的过程, 并实现这个结果。

[914]

程序6.2.1 你的第一个TOY程序

```
20: 8A15 R[A] <- M[15] load first summand into a
11: 8B16 R[B] <- M[16] load second summand into b
12: 1CAB R[C] <- R[A] + R[B] c = a + b
13: 9C17 M[17] <- R[C] store result
14: 0000 halt

15: 0008 integer value 810
16: 0005 integer value 510
17: 0000 result
```

从PC的10处开始，该程序将存储位置15和16处的两个数字相加，并将结果000D放入存储位置17

| PC | IR   | R[A] | R[B] | R[C] | M[17] |
|----|------|------|------|------|-------|
| 10 | 8A15 | 0008 |      |      |       |
| 11 | 8B16 | 0008 | 0005 |      |       |
| 12 | 1CAB | 0008 | 0005 | 000D |       |
| 13 | 9C17 | 0008 | 0005 | 000D | 000D  |
| 14 | 0000 | 0008 | 0005 | 000D | 000D  |

指令执行跟踪

内存导出

|    |      |
|----|------|
| 10 | 8A15 |
| 11 | 8B16 |
| 12 | 1CAB |
| 13 | 9C17 |
| 14 | 0000 |
| 15 | 0008 |
| 16 | 0005 |
| 17 | 000D |



0000 0000 0000 1101 ← 结果（二进制）  
0 0 0 D ← 结果（十六进制）

915

**操作机器** 我们通常可以通过键盘、显示器和触控板等 I/O 设备与计算机进行通信。在某种意义上说，TOY 也有 I/O 设备。接下来，我们描述程序员如何与像 TOY 这样的机器进行通信，以实际运行程序。左边是对我们假想的 TOY 机的前面板的描述。对应控制、输入、输出，它只有三个简单的设备：按钮、开关和指示灯。它们都是简单的开 / 关机制，但很少：只有 24 个开关和指示灯、4 个按钮。再也没有其他任何物件，没有键盘、打印机或显示器，也没有互联网连接、无线网卡、扬声器或触摸板。只有按钮来对应控制，开关对应输入，指示灯对应输出。不过，就像我们现在描述的那样，这足以操作一个基本特征相同的通用计算机来执行有价值的计算。

**按钮。**也许计算机最基本的控制就是打开或关闭电源。PDP-8 为此有一个键，TOY 有一个按钮。程序员要做的第一件事就是打开机器（最后一件事情就是把它关掉）。此外，另外三个基本功能由按钮控制：

- 将字节加载（LOAD）到计算机的内存中。

- 查看 (LOOK) 计算机内存中字的值。
- 运行 (RUN) 一个程序。

我们稍后将简短地讨论这些功能。

开关。使用这些机器的程序员用开 / 关来表示二进制值。开关处于上面位置时表示 1；处于下面位置时表示 0。TOY 计算机有两组开关：用于指定存储器位置的 8 个开关和用于指定内存字的值的 16 个开关。这些开关是 TOY 的输入设备。

指示灯。TOY 的输出设备是开关下的灯组。同样，8 个灯用于表示存储器中的位置，另外 16 个灯表示存储器中字的值。

[916]

运行一个程序。为了在 TOY 之类的机器上运行程序，程序员通常首先需要预约使用机器的时间，并在指定的时间到达机房，并提前将要执行的程序写在一张纸上。下面我们详细分析运行第一个程序所需的步骤。为简洁起见，我们通过指定开关和灯光的十六进制值来“以十六进制思考”，实际上每个开关或灯光对应一位。例如，当我们说“将 DATA 开关设置为 1CAB”时，我们的意思是“打开 DATA 开关组中对应于位串 0001110010101011 中 1 的开关”。程序员将按照下述步骤实现和运行程序 6.2.1：

- 打开机器（按 ON / OFF 按钮）。
- 将 ADDR 开关设置为 10，将 DATA 切换到 8A15，然后按 LOAD。
- 将 ADDR 开关设置为 11，将 DATA 切换到 8B16，然后按 LOAD。
- 将 ADDR 开关设置为 12，将 DATA 切换到 1CAB，然后按 LOAD。
- 将 ADDR 开关设置为 13，将 DATA 切换到 9C01，然后按 LOAD。
- 将 ADDR 开关设置为 14，将 DATA 切换到 0000，然后按 LOAD。
- 将 ADDR 开关设置为 15，将 DATA 切换到 0008，然后按 LOAD。
- 将 ADDR 开关设置为 16，将 DATA 切换到 0005，然后按 LOAD。
- 将 ADDR 开关设置为 10，然后按 RUN 按钮。
- 将 ADDR 开关设置为 17，然后按下 LOOK 按钮。
- 记下计算出的答案，如数据灯（000D）所示。
- （通常）对于其他数据值须重复前面 5 个步骤。
- 关闭机器（按 ON / OFF 按钮）。

总之，打开机器时，我们不能假定任何内存、寄存器和 PC 的值（除了 R[0] 是 0000），所以我们需要加载程序和数据（打开机器之后的七个步骤）才能运行程序。运行程序后，我们需要检查存储结果的内存位置以获得答案。

我们有必要对这个交互方式的本质进行分析。从本质上讲，程序员一次只能与机器进行一位的通信。这个过程是粗糙简陋的，但人们依旧忍耐着，因为开发程序远远优于使用铅笔和纸张、算尺或机械计算器进行计算，而且这是当时唯一可用的方法。我们很快就会看到许多短程序如何进行数值计算的例子，而通过其他方式或许很难完成这些计算任务。

当然，不久之后出现了更好的输入 / 输出设备，如键盘、打印机、纸带和磁带，但这种编程方法当然是许多科学家、工程师和学生的起点（是的，这就是 20 世纪 70 年代早期大学生学习编程的方式）。

[917]

**条件和循环** 我们按照第 1 章类似的讲述办法，学习了 TOY 程序，并学习了机器指令的数据类型和基本操作。接下来我们将转到如何控制结构上。这个过程会变得越来越有趣。

在第 1 章学到的第一个控制流结构是条件和循环。因此，我们接下来考虑用 TOY 的分支语句来实现这些结构。

举个例子，考虑如何计算两个正整数  $a$  和  $b$  的最大公约数 (gcd)。我们在 2.3 节中研究了一种基于整数除法 (有余数) 的算法。由于 TOY 没有除法指令，我们将从以下版本开始，在 Java 中的实现如下：

```
public static int gcd(int a, int b)
{
 while (a != b)
 if (b > a) b = b - a;
 else a = a - b;
 return a;
}
```

这个代码是基于一个简单的想法：如果  $b$  大于  $a$ ，任何能够整除  $a$  和  $b$  的数字 (如  $a$  和  $b$  的最大公约数) 也一定可以整除  $b-a$ ；如果  $a$  大于  $b$ ，任何能够整除  $a$  和  $b$  的数字也一定能够整除  $a-b$ 。当  $a$  和  $b$  是正数时，对于每一次的循环迭代，其中较大的数字都会减少 (但依然是正数)，直到  $a$  和  $b$  相等，这就是  $a$  和  $b$  的最大公约数，也是过程中涉及的所有数字的最大公约数。计算过程示例如右。

这就是两千多年前提出的欧几里得算法，直到今天我们依然还在学习它。这个版本的实现有时可能会很慢：例如，你可能已经注意到，在刚才的例子中  $a$  变得小于  $b$  之前，1092 减了 6 次 (其实我们可以使用除法取余数来完成相同的任务)。如果其中一个数字非常小，那么该算法可能需要的时间与另一个数字的大小成比例。例如，该算法需要 7214 次迭代，发现 7215 和 7214 的最大公约数是 1。不过请放心，比这更有效率的算法的版本已经被相当详细地研究了，即使像 TOY 一样的机器也能够高效地解决问题 (见练习 6.2.19)。

目前，我们主要关注如何实现这个函数的计算功能部分，假定程序员已经将输入数据输入到指定的内存位置，如程序 6.2.1 所示。在下一节中，我们将讨论如何将代码打包为一个函数。

我们的关键任务是有效地使用 TOY 的分支语句来实现循环和条件，如下所示。

为了实现一个 while 循环，我们把计算表达式的值的代码放在某个内存位置  $yy$ ，然后用指令  $C0yy$  (一个无条件的  $yy$  分支) 实现循环。在循环中，我们执行计算表达式的代码，当且仅当这个表达式的值是假的时候，在某个寄存器 (比如说  $R[1]$ ) 中写入 0。然后，如果寄存器为 0，我们使用条件分支  $C0xx$  将控制转移到循环之后的指令 (在地址  $xx$  处)。与上面展示的 if 语句的实现类似。

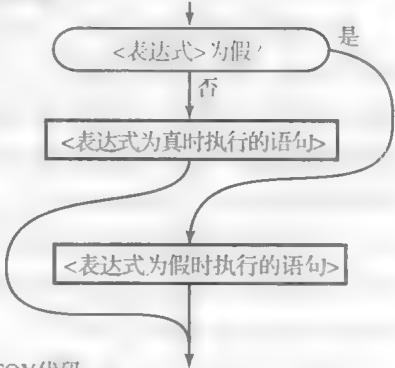
| a    | b    |
|------|------|
| 7215 | 6123 |
| 1092 | 6123 |
| 1092 | 5031 |
| 1092 | 3939 |
| 1092 | 2847 |
| 1092 | 1755 |
| 1092 | 663  |
| 429  | 663  |
| 429  | 234  |
| 195  | 234  |
| 195  | 39   |
| 156  | 39   |
| 117  | 39   |
| 78   | 39   |
| 39   | 39   |

最大公约数的计算轨迹

Java代码

```
if (<表达式>)
 <表达式为真时执行的语句>
else
 <表达式为假时执行的语句>
```

流程图



TOY代码

```
<表达式>对应的机器指令代码，当且仅当表达式为假时在R[1]中写入0
```

**Clyy** ← 如果  $R[1]$  为 0，则跳转到  $yy$

```
<表达式为真时执行的语句>
对应的机器指令代码
```

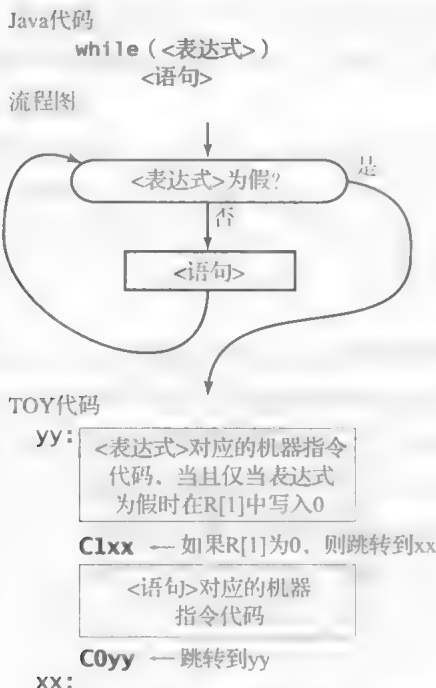
**C0zz** ← 跳转到  $zz$

```
yy: <表达式为假时执行的语句>
 对应的机器指令代码
```

zz:

实现一个条件





#### 实现一个循环

从这些结构中可以看出, Java 程序中的任何循环或条件都可以直接在 TOY 程序中实现。每行 Java 代码只对应若干个 TOY 指令。作为进一步的证据, 本节末尾的一些练习将研究在本书早期学到的 Java 程序的 TOY 指令实现。与 Java 一样, 在编程的世界里, 条件语句和循环语句只包含几条指令就可以执行数千或数百万条指令, 甚至更多。

在 TOY 的程序设计中, 我们甚至可以不局限于这几种条件和循环的实现方式。例如, 在程序 6.2.2 中, 我们实际上只用了一个条件判断, 就同时实现了 while 循环和 if 语句这两个条件语句。事实上, 程序员可以使用分支跳转指令来控制 TOY 中控制流的跳转, 而这些跳转结构可能无法自然地用条件语句和循环实现。其实最好的编程方法就是使用我们在现代编程中使用的几个构件块 (条件、循环和嵌套等), 但是人们花了很多年的时间才接受这个观点。为了表述更加清楚, 我们在本书中采用了一种折中的方法, 在编程时基本会采用刚刚分析的框架作为模板 (同时使用类似 Java 的代码作为文档), 但是当代码可以适当简化时我们也会采取一些编程的捷径。

完成了上述设计, 程序员就可以使用开关来将程序 6.2.2 的指令和数据输入到存储器中 20 到 2D 的位置上, 然后将地址开关设置为 20, 按下 RUN, 并用指示灯来显示 2D 位置的值。运行程序并观察指示灯的结果。假设程序计算 195 和 273 的最大公约数, 结果会是 39。程序员可以根据需要运行程序, 在 2B 和 2C 中输入新的数字对, 将地址开关重新设置为 20, 按 RUN 键并观察 2D 中的结果。

跟踪轨迹是开发和调试 TOY 程序时更加有效的好方法。首先, 程序将 195 (00C3) 加载到 R[A] 中, 将 273 (0111) 加载到 R[B] 中。然后将它们相减, 把结果 -78 (FFC3) 写入 R[C]。由于这个值不是零, 进入循环, 然后测试差值是否为正。由于它是负的, 从 R[B] 中减去 R[A], 在 R[B] 中留下 78 (004E), 然后返回循环的另一次迭代。在循环的下一次迭代中, 从 R[A] 中减去 R[B], 在 R[A] 中保留 117 (0075)。然后再次从 R[A] 中减去 R[B], 在 R[A] 中留下 39 (0027)。循环的最后一次迭代是从 R[B] 中减去 R[A], 在两个寄存器中都

留下结果 39 (0027)。

我们可以想象，这个经典的算法使得早期程序员通过不断的试验认识到对于计算机来说计算可以如此简单。是否有人可以用笔完成这些计算呢？通过这种方式实现了更快的计算，这使得一些数学家和科学家成为最初的程序员。他们不断地追求更快的算法，并致力于开发更有效的算法，直到今天。

920

程序6.2.2 条件和循环：欧几里得算法

|          |                         |                |
|----------|-------------------------|----------------|
| 20: 8A2B | R[A] <- M[2B]           | a = p          |
| 21: 8B2C | R[B] <- M[2C]           | b = q          |
| 22: 2CAB | R[C] <- R[A] - R[B]     | while (a != b) |
| 23: CC29 | if (R[C] == 0) PC <- 29 | {              |
| 24: DC27 | if (R[C] > 0) PC <- 27  | if (b > a)     |
| 25: 2BBA | R[B] <- R[B] - R[A]     | b = b - a      |
| 26: C022 | PC <- 22                | else           |
| 27: 2AAB | R[A] <- R[A] - R[B]     | a = a - b      |
| 28: C022 | PC <- 22                | }              |
| 29: 9A2D | return a (= b = GCD)    | return a       |
| 2A: 0000 | halt                    |                |

|          |                                 |   |
|----------|---------------------------------|---|
| 2B: 00C3 | integer value 195 <sub>10</sub> | p |
| 2C: 0111 | integer value 273 <sub>10</sub> | q |
| 2D: 0000 | result                          |   |

PC从20开始，程序从内存位置2B和2C中读取两个数字，计算它们的最大公约数，并将结果放在内存位置2D中。

| PC | IR   | R[A] | R[B] | R[C] |
|----|------|------|------|------|
| 20 | 8A2B | 00C3 |      |      |
| 21 | 8B2C | 00C3 | 0111 |      |
| 22 | 2CAB | 00C3 | 0111 | FFC3 |
| 23 | CC29 | 00C3 | 0111 | FFC3 |
| 24 | DC27 | 00C3 | 0111 | FFC3 |
| 25 | 2BBA | 00C3 | 004E | FFC3 |
| 26 | C022 | 00C3 | 004E | FFC3 |
| 27 | 2AAB | 0075 | 004E | 0075 |
| 28 | C022 | 0075 | 004E | 0075 |
| 29 | 9A2D | 0075 | 004E | 0075 |
| 2A | 0000 | 0075 | 004E | 0075 |



0000 0000 0010 0111 ← 结果 (二进制)  
0 0 2 7 ← 结果 (十六进制)

921

**存储程序计算** TOY 计算机的基本特征之一是它将计算机程序存储为数字，并且数据和程序都存储在相同的主存储器中。这是理解计算的基本性质的关键，也是经过曲折历史后的深刻教训。

**数据作为指令和指令作为数据。**如程序 6.2.3 展现的这一基本特征，它实现了一个数字序列的求和操作。该序列可以是以 0000 结尾的任意长度。该程序的操作轨迹如程序右图所示，仔细观察 (轨迹中省略了 R[1] 和 M[1B]，因为它们中的每一个值只改变一次)。计算开始时和程序 6.2.1 基本相同，后面就有很大差异了。前两个数字相加并将结果留在 R[A] 中后，程序将位置 12 的指令 8B1D 加载到 R[D] 中，再加上 1，然后将结果 8B1E 存回到位置 12 中。然后，跳转到位置 12，该指令用于向 R[B] 中加载下一个要加到 R[A] 中的数字，如

此继续循环，直到在数据中遇到 0000 时停止。

程序6.2.3 自修改代码：计算一串数字的和

```
10: 7101 R[1] <- 0001
11: 8A1C R[A] <- M[1C] load first number into a
12: 881D R[B] <- M[1D] while (b != 0)
13: CB19 if (R[B]==0) PC <- 19 {
14: 1AAB R[A] <- R[A] + R[B] a = a + b
15: 8D12 R[D] <- M[12] modify instruction at M[12]
16: 1D1D R[D] <- R[D] + 1 to load next number into
17: 9D12 M[12] <- R[D] b on next iteration
18: C012 PC <- 12 }
19: 9A1B store result
1A: 0000 halt

1B: 0000 result
1C: 0001 data
1D: 0008 integer value 810
1E: 001B integer value 2710
1F: 0040 integer value 6410
20: 0000
```

PC从10开始，该程序将存储在位置1C到1F（以0000结尾）的数字序列相加并将结果存储在位置1B。



0000 0000 0010 0111 ←结果（二进制）  
0 0 2 7 ←结果（十六进制）

| PC | IR   | R[A] | R[B] | R[D] | M[12] |
|----|------|------|------|------|-------|
| 10 | 7101 |      |      |      | 881D  |
| 11 | 8A1C | 0001 |      |      | 881D  |
| 12 | 881D | 0001 | 0008 |      | 881D  |
| 13 | CB19 | 0001 | 0008 |      | 881D  |
| 14 | 1AAB | 0009 | 0008 |      | 881D  |
| 15 | 8D12 | 0009 | 0008 | 881D | 881D  |
| 16 | 1D1D | 0009 | 0008 | 881E | 881D  |
| 17 | 9D12 | 0009 | 0008 | 881E | 881E  |
| 18 | C012 | 0009 | 0008 | 881F | 881E  |
| 12 | 881E | 0009 | 001B | 881E | 881E  |
| 13 | CB19 | 0009 | 001B | 881E | 881E  |
| 14 | 1AAB | 0024 | 001B | 881E | 881E  |
| 15 | 8D12 | 0024 | 001B | 881E | 881E  |
| 16 | 1D1D | 0024 | 001B | 881F | 881E  |
| 17 | 9D12 | 0024 | 001B | 881F | 881F  |
| 18 | C012 | 0024 | 001B | 881F | 881F  |
| 12 | 881F | 0024 | 0040 | 881F | 881F  |
| 13 | CB19 | 0064 | 0040 | 881F | 881F  |
| 14 | 1AAB | 0064 | 0040 | 881F | 881F  |
| 15 | 8D12 | 0064 | 0040 | 881F | 881F  |
| 16 | 1D1D | 0064 | 0040 | 8820 | 881F  |
| 17 | 9D12 | 0064 | 0040 | 8820 | 8820  |
| 18 | C012 | 0064 | 0040 | 8820 | 8820  |
| 12 | 8820 | 0064 | 0000 | 8820 | 8820  |
| 13 | CB19 | 0064 | 0000 | 8820 | 8820  |
| 19 | 9A1B | 0064 | 0000 | 8820 | 8820  |
| 1A | 0000 |      |      |      |       |

这样的代码被称为自修改代码（self-modifying code）。我们之所以引入这个程序，是因为它简洁地说明了存储程序计算的基本概念。内存位置 12 的内容是指令还是数据？其实是两个都是！当我们加 1 时，就是数据；当 PC 引用它并加载到 IR 时，它就是一条指令。由于程序和数据共享相同的内存，机器可以在执行时修改其数据或程序本身。也就是说，代码和数据是相同的，或者至少可以是相同的。当使用程序计数器引用存储器中的内容时它就是指令，当使用指令引用时它就是数据。

这样的自修改代码在现代计算中很少使用，因为它很难被理解、调试和维护。我们将在下一节中分析另一种数字序列求和的方案。但是将指令当作数据来处理的能力在计算中是非常重要的，正如下一章和本章的其余部分所讨论的那样。

一些影响。将程序作为数据来处理的能力在现代计算基础架构中至关重要：

- 任何应用程序（application）在下载或安装时都被视为数据，但在启动时将其视为程序。

- 编译器 (Compiler) 程序的功能如下：它读取其他程序作为输入数据，将其生成机器语言并输出。所有的编程语言都基于编译器工作。
- 现代云计算 (cloud computing) 基于虚拟机 (virtual machine) 的概念，其中一台计算机运行基于另一台计算机的程序。事实上，TOY 本身就是一个虚拟机器，我们将在 6.4 节中进行分析。

这些只是一些例子，我们将在本章中重新审视这个概念。

将程序视作数据并非没有漏洞。例如，计算机病毒是通过编写新程序或修改现有程序而传播的 (恶意) 程序。正如在第 5 章中学习的，图灵理论已经告诉我们，一般来说，没有有效的方法来区分恶意病毒、有用的应用程序或数据。这个缺点是存储程序模型不可避免的后果。我们将在下一节中针对 TOY 计算机的环境分析一个具体的示例。

**冯·诺依曼机器** 如前面所述，到 20 世纪四五十年代，科学家和工程师们进行了大量的计算，它们不仅仅用于战争，如弹道学、核武器和密码学，而且还用于和平时期的太空飞行和气象学等。电子元件比机械元件更快的想法更加根深蒂固。

但是，许多早期计算机的实现方式是模拟机械计算器。运营商必须通过插入电缆和设置交换机组来“编程”计算机，这很烦琐、耗时且容易出错。所有内存都被用于存储数据。例如 ENIAC 就是这样的一台计算机，它在 20 世纪 40 年代中期由 Eckert 和 Mauchly 在宾夕法尼亚大学开发 (ENIAC 被认为是世界上第一台电子计算机——译者注)。

与此同时 (实际上早在 20 世纪 30 年代)，数学领域也对此具有极大的兴趣，因为图灵发现了我们在第 5 章中分析过的巧妙的理论结构。图灵的工作帮助我们更加深刻地理解了计算本质特性。

普林斯顿学者冯·诺依曼是 Eckert 和 Mauchly 二人在 ENIAC 项目上的顾问，也是这个项目的继任者。当时他对这个机器的两方面的应用充满了浓厚的兴趣：一个是弹道学计算；一个是原子弹开发中需要的精确计算。

1945 年，冯·诺依曼在从普林斯顿到洛斯阿拉莫斯的列车上，写下了他对 ENIAC 进行改进的报告，这就是 EDVAC 报告的初稿。在这份备忘录中完整描述了存储程序计算模型。由于冯·诺依曼是普林斯顿大学的教授，而图灵是该校的研究生，所以他受到了图灵思想的影响，而且用一种独特的方法将图灵理论与 Eckert 和 Mauchly 面临的实际挑战结合了起来。冯·诺依曼抵达洛斯阿拉莫斯后不久，一位名叫赫尔曼·戈德斯坦 (Herman Goldstine) 的少尉意识到这个想法会引起人们的浓厚兴趣，并且广泛地传阅了这份备忘录。世界各地的科学家立即看到了存储程序模型的价值，基于该模型的计算机 (几乎所有的计算机都基于这个模型) 都被称为冯·诺依曼机器。许多历史学家认为，Eckert 和 Mauchly 应该得到这一殊荣 (就像图灵一样)，但无疑是冯·诺依曼的备忘录使得这一构思在世界各地生根发芽。

存储程序模型使计算机能够执行任何类型的计算，而不需要用户做物理上的改变或重新配置硬件。自从冯·诺依曼第一次阐述它以来，这个简单且基本的模型几乎已经被用于所有的计算机中。

现在看来，冯·诺依曼的架构显而易见。然而，当时有许多研究小组正在朝不同的方向工作，而到底是使用存储程序模型构



冯·诺依曼 (1903—1957)

建计算机还是应该构建能够重新布线和重新配置的计算机仍存在争议。事实上，图灵的理论表明，只要一台计算机的基本指令集足够丰富（TOY 计算机的指令就已经足够），那么无论在物理上再怎么改装或者重配置，都不能使它解决更多问题。除图灵之外，冯·诺依曼是世界上为数不多的同意这一观点的人之一。他将一次火车旅途中的偶然发现完整地表达出来，而这改变了整个世界。

925

## 问答环节

问：程序员真的会通过拨动开关来输入程序吗？

答：是的。很多人都学会了这个技巧。即使有更好的输入/输出设备可用，也需要通过开关来输入其中一个设备的驱动程序。

问：寄存器和内存字有什么区别？

答：它们都可以存储 16 位整数，但它们在计算机内扮演不同的角色。内存的目的是保存程序和数据——我们希望内存尽可能大。寄存器的目的是提供一个中转存储，以便从 ALU 获取数据或者向 ALU 发送数据，我们只需要有限数量的寄存器。通常情况下，计算机使用更昂贵技术的寄存器，因为它们几乎在每个指令都会被用到。计算机中的寄存器数量是一个设计决策。

问：专用计算机或专用微处理器今天还在制造吗？

答：是的，因为在硬件上实现，可以做得比在软件中更快地完成简单的事情。

问：很难想象会存在没有循环语句和条件语句的编程语言。人们真的那样编程过吗？

答：当然。程序员用流程图设计他们的逻辑，而早期的高级语言有一个“goto”语句，它可以直接转换成机器语言的分支跳转语句。学术界很早就出现了使用循环、条件和函数的“结构化编程”思想，但直到 20 世纪 70 年代才被许多程序员认真对待。一个著名的转折点是 E.W.Dijkstra 在 1968 年给《ACM 通信》(the Communications of the ACM) 编辑的一封信，题目是《Goto 被认为是有害的》(Goto considered harmful)。在这封信中，他主张在所有高级编程语言中废除 goto 语句，因为使用它的程序很难理解、调试和维护。这个观点用了十年才被普遍接受，而结构化编程自此以后被认为是理所当然的。

926

## 练习

6.2.1 TOY 的存储空间有多少位？计上所有的寄存器（包括 PC）和主存储器。

6.2.2 TOY 使用 8 位内存地址，这意味着内存可以有 256 个字长。我们可以使用 32 位地址处理多少个字长的内存？64 位地址呢？

6.2.3 假设我们要使用与 TOY 相同的指令格式，但是使用 32 位地址。我们需要什么字长？描述使用这个设计方案可能会出现的问题。

6.2.4 给出一条指令，将程序计数器更改为内存地址 15，而不管任何寄存器或内存单元的内容如何。

答案：C015 或 F015。两条指令都依赖于 R[0] 始终为 0000 的事实。

6.2.5 列出七条指令（都有不同的操作码），将 0000 放入寄存器 A。

答案：1A00, 2Axx, 3A0x, 4Axx, 5A0x, 6A0x, 7A00，其中 x 是任何十六进制数字。

6.2.6 列出三种方式（不同的操作码）将程序计数器 PC 设置为 00，而不改变任何寄存器或存储单元的内容。

答案：C000, E0xy, F000。

- 6.2.7 列出五个指令（全部具有不同的操作码），它们的功能是空操作。排除第二位数字为 0 的情况。  
答案：1xx0, 1x0x, 2xx0, 3xxx, 5xx0, 6xx0 或 D0xx，其中 x 是除 0 之外的任何十六进制数字。
- 6.2.8 列出六种方法将 R[B] 的内容赋值给 R[A]。  
答案：1AB0, 1A0B, 2AB0, 3ABB, 4A0B, 4AB0, 5AB0 和 6AB0。
- 6.2.9 TOY 中没有支持非负判断的分支语句。请说说如何实现如果 R[A] 大于或等于 0，则跳转到内存地址 15。  
答案：按顺序使用正数判断分支语句和值为 0 的判断分支语句：CA15 DA15。
- 6.2.10 填写本表格中的空格：

927

| 二进制              | 十六进制 | TOY指令                   |
|------------------|------|-------------------------|
| 0001001000110100 | 1234 | R[2] <- R[3] + R[4]     |
| 1111111111111111 | FFFF |                         |
| 1111101011001110 | FACE |                         |
| 0101011001000100 | 5644 |                         |
| 1000000000000001 | 8001 |                         |
| 0101000001000011 | 5043 |                         |
| 0001110010101011 | 1CAB |                         |
|                  |      | R[F] <- R[F] & R[F]     |
|                  |      | R[8] <- M[88]           |
|                  | 7777 |                         |
|                  |      | if (R[C] == 0) PC <- CC |

- 6.2.11 TOY 中没有绝对值函数。给出一系列 TOY 指令，将 R[s] 的绝对值存储为 R[d]。
- 6.2.12 TOY 中没有按位 NOR 操作符。给出一组三个 TOY 指令的序列，当且仅当 R[s] 和 R[t] 相应位中的一个或两个为 0 时，将 R[d] 的对应位设置为 1。
- 6.2.13 TOY 中没有按位 OR 运算符。给出一组三个 TOY 指令的序列，当且仅当 R[s] 和 R[t] 相应位中的一个或两个为 1 时，将 R[d] 的每个对应位设置为 0。  
答案：3DAB 4EAB 1CDE。
- 6.2.14 TOY 中没有按位 NAND 运算符。给出一组三个 TOY 指令的序列，当且仅当 R[s] 和 R[t] 中的相应位都是 1 时，将 R[d] 的相应位设置为 0。
- 6.2.15 TOY 中没有按位 NOT 运算符。给出一组三个 TOY 指令的序列，将 R[d] 的每个位设置为 R[s] 中对应位值的相反数。  
答案：7101 2B01 4BAB 或 7101 2B01 2BBA。

928

- 6.2.16 证明减法运算符是多余的。也就是说，解释如何在不使用操作码 2 的 TOY 指令序列来计算  $R_d = R_s - R_t$ 。
- 6.2.17 16 个 TOY 指令中哪个用不全这 16 位？  
答案：停机（仅使用前 4 位），间接加载（不使用第 3 个十六进制数字），间接存储（不使用第 3 个十六进制数字），跳转寄存器（不使用两个十六进制数字）
- 6.2.18 根据 TOY 的二进制补码的规则，我们会把一些整数解释为负数。这种情况对于哪些指令会有影响？  
答案：判断为正的分支指令将 0001 和 7FFF 之间的整数视为正数。右移指令是一个算术移位，所以如果最左边的位是 1，则空出的位置用 1 来填充。所有其他指令（甚至减法！）并不关心 TOY 是否具有负整数。

6.2.19 按照下面的方法修改 while 循环，以改进文本中给出的最大公约数算法在 TOY 上的实现。这里需要增加一个初始化为 0 的变量 *c*，并对循环做如下修改：

- 如果 *a* 和 *b* 是偶数，则  $\text{gcd}(a, b) = 2 \text{gcd}(a/2, b/2)$ ，所以将 *a* 和 *b* 除以 2 并将 *c* 增加 1。
- 如果 *a* 是偶数并且 *b* 是奇数，则  $\text{gcd}(a, b) = \text{gcd}(a/2, b)$  将 *a* 除以 2。
- 如果 *a* 是奇数，*b* 是偶数，则将 *b* 除以 2（道理同上）。
- 否则，*a* 和 *b* 都是奇数，按照以前的方式进行（即用它们的差取代两者中更大的一个）。

最后，将计算得到的结果左移 *c* 位，用于补偿在计算过程中略去的因子 2。设计一个客户程序，它从标准输入读入两个整数，并将计算出的最大公约数打印到标准输出。（分析表明，该算法的最坏情况运行时间是输入数据位数的平方级——这比正文中给出的版本快得多。分析的细节超出本书讨论的范围，我们不再展开。）

929

## 6.3 机器语言编程

在本节中，我们将继续描述如何在 TOY 中实现我们在本书前面所学习的 Java 语言机制，并完成一些 TOY 计算机上有趣的编程任务。我们的主要目标是让你相信，TOY 编程与 Java 编程一样有趣、令人满意，TOY 比想象的要强大得多。

具体来说，我们会学习如何在 TOY 的程序中实现函数、数组、标准输入输出和链接结构，也就是我们前面学过的编程的基本构建块。理论上，这些结构表明可以开发与我们编写的任何 Java 程序相对应的机器语言程序。事实上也确实是这样的，因为 Java 编译器就是这样做的。

当然，在资源方面可能会存在限制。我们真的能用 4096 位的内存完成实际的计算任务吗？本节的目标之一就是说服你，我们当然可以。不过，技术的进步已经使这种限制变得很小。在 6.4 节中，我们将讨论 TOY 的扩展，使它在这方面看起来更像是一个现代计算机，但不会改变编程模型。在这样的机器上，你当然可以编写 TOY 程序来执行任何计算任务，就像你编写 Java 程序在你的计算机上执行的计算一样。

在实践层面上，你会发现用机器语言实现各种任务都是切实可行的。实际上，许多早期的应用程序就是这样实现的，而且这种状况维持了很多年，因为当时使用高级语言的性能损失太大了。事实上，大多数这样的代码是用汇编语言（assembly language）编写的，汇编语言类似于机器语言，只是它允许操作码、寄存器和内存位置使用特定的符号名称（见练习 6.4.13）来表示。在 20 世纪 70 年代，用汇编语言写就的、可以扩展到数以万行计的代码并不罕见。所以我们也是在回顾历史，但即使是现在，人们也会为性能关键的应用程序编写这样的代码。

930

最重要的是，我们的目标是让你了解计算机在运行程序时具体做了什么。

**函数** 在条件和循环之后，我们在第 2 章中学习的下一个控制流结构就是函数（function）。TOY 的跳转指令是为此目的而设计的。实现函数的方式有很多种，由于我们的主要目标是展示这个概念，所以我们选择其中最简单的一种方法来进行。要实现一个函数，我们必须：

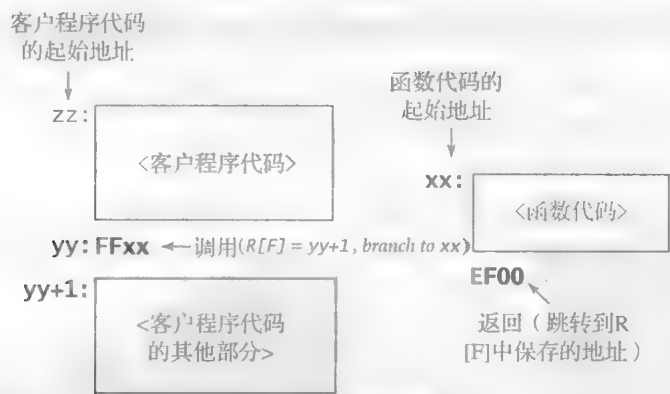
- 将控制流转移到该函数；
- 将客户程序的参数传递给函数；
- 从函数返回一个值给客户程序；
- 将控制权交还给客户程序。



我们选择使用寄存器来帮助程序完成这些任务。对于欧几里得算法，我们按照以下步骤进行：

- 使用 TOY 的跳转和链接（jump and link）指令来将控制权交给函数。该指令还将返回地址（客户程序中下一条指令的存储器地址）保存在一个寄存器中（我们使用 R[F]）。
- R[A] 和 R[B] 用于存储参数和返回值。
- 使用 TOY 的寄存器跳转（jump register）指令将控制权返回给客户程序。具体而言，指令 EF00 用于将 PC 设置为 R[F] 中保存的返回地址。

一个函数调用的典型控制流程如下图所示。



一个函数调用的实现

例如，通过上面的方法，很容易将程序 6.2.2 中的代码转换为计算 R[A] 和 R[B] 的最大公约数（GCD）的函数，将结果留在 R[A] 和 R[B] 中：将 M[29] 更改为寄存器跳转指令 EF00，以便客户程序可以使用跳转和链接指令 FF22 来调用该函数。这段代码实现如下。

一个计算GCD的函数

|     |      |                         |                     |
|-----|------|-------------------------|---------------------|
| 22: | 2CAB | R[C] <- R[A] - R[B]     | c = a - b           |
| 23: | CC2C | if (R[C] == 0) PC <- 2C | while (a != b)      |
| 24: | DC2A | if (R[C] > 0) PC <- 2A  | { if (b > a)        |
| 25: | 2BBA | R[B] <- R[B] - R[A]     | b = b - a           |
| 26: | C022 | PC <- 22                | else                |
| 27: | 2AAB | R[A] <- R[A] - R[B]     | a = a - b           |
| 28: | C022 | PC <- 22                | }                   |
| 29: | EF00 | PC <- R[F]              | return              |
|     |      |                         | [R[A] = R[B] = GCD] |

931

程序 6.3.1 是使用这个 GCD 函数来测试一个整数是否为素数的客户程序（与前面的问题一样，如果我们有除法指令的话，这个任务会容易很多！）。方法很简单：当且仅当它和每个比它小的正整数的最大公约数都是 1 时，这个数字就是素数（如果一个数字有大于 1 的约数，这个数字和约数的最大公约数是约数本身）。与程序 2.3.1 一样，我们可以在到达最小约数的上限后停止计算。由于我们没有平方根函数，所以我们只能使用 255 作为这个上限，因为  $255^2$  大于任何用补码形式表示的正的 16 位二进制数（或者，我们可以用一些其他方法很容易地计算给定数字的平方根的一些上限）。

程序下面显示函数调用之前和之后，以及达到停机指令时 R[9]、R[A] 和 R[B] 的值。

请注意，该函数使用 R[C]，因此调用程序不能期望 R[C] 在函数调用之后有具与调用之前相同的值。而且，函数当然不能使用 R[9]，因为调用程序在那里保存了一个索引，也不能使用 R[F]，因为返回地址保存在那里。在资源如此稀缺的情况下，这种程序中的“契约”在这个层面的编程中起着重要的作用，我们在后面会经常遇到。

与条件和循环一样，我们可以在 TOY 中实现一些在 Java 中不是很方便实现的函数调用机制。例如，我们可能使用多个寄存器作为返回值。在一些情况下，一些程序员在调用函数之前会保存所有寄存器；也存在完全相反的情况，一些程序员会要求每个函数负责保存所有寄存器的值，然后再继续执行，在从函数返回时将其恢复到原始值。现代函数调用机制倾向于后一种方法。

程序 6.3.1 中使用的函数调用机制无法实现函数递归，但很容易使用堆栈开发更简单的机制（参见练习 6.3.27）。像往常一样，我们的目的不是要覆盖所有的细节，而只是为了说服你，我们在 TOY 这样的机器上实现在 Java 中的基本构造并不是很难。

程序6.3.1 调用函数：检测素数

```
30: 7101 R[1] <- 0001
31: 75FF R[5] <- 255
32: 7901 R[9] <- 0001 i = 1
33: 2C59 R[C] <- R[5] - R[9] while (i < 255)
34: CC3B if (R[C] == 0) PC <- 3B {
35: 1991 R[9] <- R[9] + R[1] i = i + 1
36: 1A09 R[A] <- R[9] a = i
37: 8B3D R[B] <- M[3D] b = p
38: FF22 R[F] <- PC; PC <- 22 a = b = gcd(a, b)
39: 2AA1 R[A] <- R[A] - R[1] if (gcd(a, b) != 1) break
3A: CA36 if (R[A] == 0) PC <- 36 }
3B: 9B3E M[3E] <- R[B] x = 1 iff p is prime
3C: 0000 halt

3D: 005B integer value 9110 p
3E: 0000 result x
```

PC从30开始，程序测试M[3D]中的数字p是否为素数。如果是，则结果为1，否则为大于1的最小约数。程序使用前面的gcd()函数，这段程序由程序6.3.2修改而来，用指令FF22调用它。

| PC | IR   | R[A] | R[B] | R[9] | M[3E] |
|----|------|------|------|------|-------|
| 37 | 8B3D | 0002 | 005B | 0002 | 0000  |
| 38 | FF22 | 0001 | 0001 | 0002 | 0000  |
| 37 | 8B3D | 0003 | 005B | 0003 | 0000  |
| 38 | FF22 | 0001 | 0001 | 0003 | 0000  |
| 37 | 8B3D | 0004 | 005B | 0004 | 0000  |
| 38 | FF22 | 0001 | 0001 | 0004 | 0000  |
| 37 | 8B3D | 0005 | 005B | 0005 | 0000  |
| 38 | FF22 | 0001 | 0001 | 0005 | 0000  |
| 37 | 8B3D | 0006 | 005B | 0006 | 0000  |
| 38 | FF22 | 0001 | 0001 | 0006 | 0000  |
| 37 | 8B3D | 0007 | 005B | 0007 | 0000  |
| 38 | FF22 | 0007 | 0007 | 0007 | 0007  |
| 3B | 0000 | 0006 | 0007 | 0007 | 0007  |

当PC在37、38、3B时的追踪

| i | gcd(i, 91) |
|---|------------|
| 2 | 1          |
| 3 | 1          |
| 4 | 1          |
| 5 | 1          |
| 6 | 1          |
| 7 | 7          |

这个例子说明我们一直在讨论的 Java 模块化编程的优势也适用于 TOY 程序。一旦我们用于计算 GCD 或用于测试素数的代码已经被调试，我们就可以在其他程序中使用它，可以用单个 TOY 指令访问。这种能力使得迅速提高机器语言编程的抽象层开发变得可行，并发

展出了我们今天仍然使用的软件基础设施的许多方面。

**标准输出** 当然，计算机发明之后最早取得的进展之一是与计算机进行通信的更好方式。我们使用了大量不同的设备，而不再只是使用开关和指示灯。我们为 TOY 选择了一种仅含有基本要素的通信手段：打孔纸带（punched paper tape）。与 TOY 本身一样，对你而言这种方法可能看起来非常粗糙，但它被广泛使用了至少十年。

打孔纸带是一种简单的介质，以非常明显的方式编码二进制数字。对于 TOY，我们使用的编码非常类似于我们已经提到过的旧版 PDP-8 计算机上使用的编码。每个 16 位二进制数被编码在纸带上连续的两行上，其中每行可以编码八位，在对应于 1 的位置上打孔（对应于 0 的位置上没有孔）。沿着纸带的中心是一系列规则间隔的小孔，过去通过这些小孔和齿轮之间的行进来拉动纸带。纸带打孔机将从计算机中取出一个 16 位的二进制字，对与该字对应的孔位进行打孔（两行），并推进纸带准备打孔下一个字。程序员可以编写一个程序在纸带上打出信息，然后查看纸带（或者如我们将看到的，稍后将其反馈到机器中），而不是使用指示灯和开关。



真实的和想象中的纸带

我们如何指示 TOY 计算机在纸带上输出一个字？这个问题的答案很简单：我们为此预留了内存位置 FF，当我们连接硬件后，每当程序在该位置存储一个字时，纸带打孔器就会被激活以在纸带上打出该字的内容。

程序 6.3.2 是一个例子，用于计算斐波那契数字并在纸带上打印出它们。计算很简单：我们在 R[A] 和 R[B] 中保存前两个斐波那契数，其中 R[A] 初始化为 0，R[B] 初始化为 1。然后我们进入一个循环，通过将 R[A] 和 R[B] 相加的结果记为 R[C]，以计算出下一个斐波那契数字；然后将 R[B] 复制到 R[A]，R[C] 复制到 R[B]。每次循环时，我们都执行指令 9AFF，它将在纸带上打印出 R[A] 的内容。结果输出纸带显示在程序下面。如果将纸带的内容与其右侧的轨迹进行比较，可以在纸带上读到斐波那契数字的二进制形式，就像对 TOY 进行编程的人读取数据那样。

请注意，TOY 程序没有明确提到纸带；它只是执行 9AFF 指令，这一简单的概念使得我们可以在不必改变 TOY 程序的基础上，用不同的输出设备（可能是电传打印机或纸带设备）替换纸带打孔机。这样的设计就是标准输出抽象的先驱，我们到现在还在使用这个理念。

程序6.3.2 标准输出：斐波那契数列

```

40: 7101 R[1] <- 0001
41: 7A00 R[A] <- 0000 a = 0
42: 7B01 R[B] <- 0001 b = 1
43: 894C R[9] <- M[4C] i = n
44: C94B if (R[9] == 0) PC <- 4B while (i > 0) {
45: 9AFF R[A] to stdout print(a)
46: 1CAB R[C] <- R[A] + R[B] c = a + b
47: 1AB0 R[A] <- R[B] a = b
48: 1BC0 R[B] <- R[C] b = c
49: 2991 R[9] <- R[9] - 1 i = i - 1
4A: C044 PC <- 44 }
4B: 0000 halt

```

4C: 000C integer value 12<sub>10</sub> . . . n

PC从40开始，程序在标准输出上写入前*n*个非零斐波那契数，其中*n*是内存位置4C的整数值。

|      | R[A] | R[B] | R[C] | R[9] |
|------|------|------|------|------|
| 0000 | 0000 | 0001 |      | 000C |
| 0001 | 0001 | 0001 | 0001 | 000B |
| 0001 | 0001 | 0002 | 0002 | 000A |
| 0002 | 0002 | 0003 | 0003 | 0009 |
| 0003 | 0003 | 0005 | 0005 | 0008 |
| 0005 | 0005 | 0008 | 0008 | 0007 |
| 0008 | 0008 | 000D | 000D | 0006 |
| 000D | 000D | 0015 | 0015 | 0005 |
| 0015 | 0015 | 0022 | 0022 | 0005 |
| 0022 | 0022 | 0037 | 0037 | 0003 |
| 0037 | 0037 | 0059 | 0059 | 0002 |
| 0059 | 0059 | 0090 | 0090 | 0001 |
| 0090 | 0090 | 00E9 | 00E9 | 0000 |

在PC = 44上追踪

输出纸带

值得注意的是，纸带实现了标准输出最重要的特征之一：对纸带的长度没有限制。这种额外的功能使得我们可以编写出产生无限量输出的程序。这个能力不仅具有实际意义（即使TOY是一个小机器，它可以进行大量的计算并产生大量的输出），而且它对计算理论也有着深远的影响，正如我们在第5章中学习的那样。

**标准输入** 当然，从纸带读取信息的设备与在纸带打孔的设备同时出现。一面是光源，另一面是16个传感器，可以快速地从纸带上读取两行，用作一个二进制字的值，1对应于打孔，0对应于无孔。同样，与纸带中心的小孔匹配的齿轮将把纸带拉到准备读下一个字的位置。

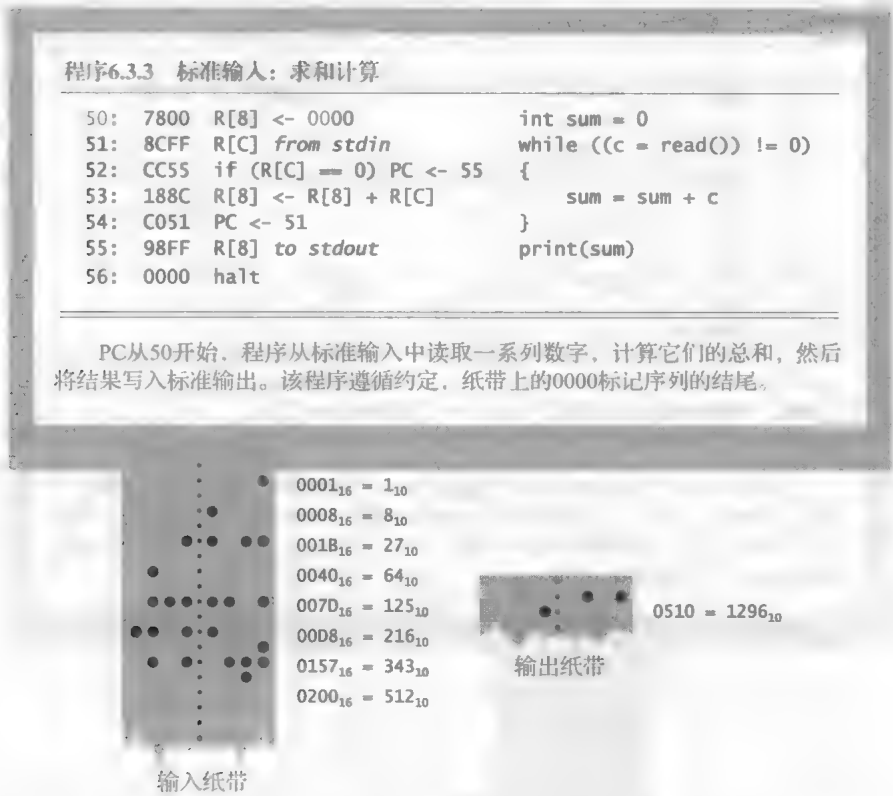
要从输入纸带读取一个字，你可能已经猜到我们会再次使用内存位置FF来达到此目的。在这种方案中，我们连接硬件，使得每当程序从该位置加载一个字时，纸带打孔器被激活以从纸带读取16位并将它们加载到指定的寄存器。

有了标准输入的支持，使用TOY进行数据处理则非常简单，如程序6.3.3所示。只需要7个TOY指令，我们就可以计算输入纸带上的数字总和。下面所示的例子证实了这一点：

$$1 + 8 + 27 + 64 + 125 + 216 + 343 + 512 = 1296$$

这个程序显然可以泛化到处理输入数据的各种计算，而这样的计算机在过去被大量使

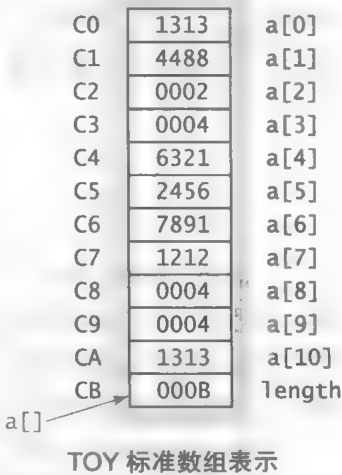
用。事实上，一台机器装载一个小程序并不奇怪，然后它就可以交由操作员简单地设置纸带，运行程序，并整天在输出纸带上收集输出。能够有效处理数据所带来的价值毋庸置疑。需要指出的是，输入数据的数量可能是无限的，这是一个反映计算理论本质的深刻概念。



**数组** 能够批量读取大量数据的能力导致需要将数据保存在存储器中以对其进行处理。当然，这带来了我们在 TOY 中的第一个数据结构——数组 (array)。我们使用一种很自然地数组表示方式，就像我们先前在 Java 中描述的那样，将数组条目存储在一个连续的内存字序列中，但是我们将该长度存储在数组末尾而不是开始位置，如右图所示。我们用那个表示长度的字的地址来表示这个数组。我们习惯于将长度放在内存区域开头的位置，但是这种方法在原理上并没有什么差异：它保留了一个基本特征，即我们可以通过将一个 a[0] 的地址加上 i 来计算出 a[i] 的地址，并且可以很容易地从数组地址中计算出 a[0] 的地址（减去长度）。

TOY 间接加载 (load indirect) 指令和间接存储 (store indirect) 指令的主要目的之一就是支持数组处理。我们将 a[i] 的地址保存在一个寄存器中。然后，为了加载/存储数组值中第 n 位值，我们使用间接加载/存储 (load/store indirect) 指令，将它加载到指定的寄存器中。

程序 6.3.4 说明了将纸带内容加载到数组中的过程。它被封装为一个函数，从纸带中读取数组中的字数量和要存储的地址，然后进入一个简单的循环，读取每个字，并将结果存储在 a[i] 中，然后通过维护索引 i 以跟踪下一个数组元素的输入。在读取和存储了所有内容之

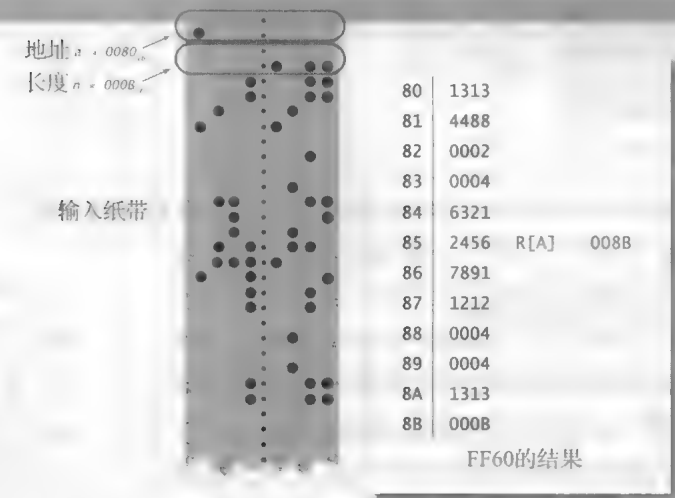


后，i 的值就是数组的长度——在停机指令前将数组长度存储在数组的末尾。返回值 R[A] 中保存的是长度变量，也就是数组末尾一项的地址（TOY 标准格式）。

程序6.3.4 数组处理：读取数组

```
60: 8AFF R[A] from stdin a = address of a[0]
61: 8BFF R[B] from stdin n = read()
62: 7101 R[1] <- 0001
63: 7900 R[9] <- 0000 i = 0
64: 22B9 R[2] <- R[B] - R[9] while (i < n)
65: C26B if (R2 == 0) PC <- 6B {
66: 1CA9 R[C] <- R[A] + R[9]
67: 8DFF R[D] from stdin
68: BD0C M[R[C]] <- R[D] a[i] = read()
69: 1991 R[9] <- R[9] + 1 i = i + 1
6A: C064 PC <- 64 }
6B: 1AA9 R[A] <- R[A] + R[9] address of a[] (TOY standard)
6C: BB0A M[R[A]] <- R[B] a.length = n
6D: EF00 PC <- R[F] return
```

PC从60开始，该程序用于从打孔纸带读取一个数组，并以TOY标准格式存储。具体地说，它从标准输入中读取地址a和整数n，并从标准输入中读取n个整数，并将其存储在存储位置M[a]、M[a+1]、...、M[a+n-1]中。然后它将n存储在M[a+n]中（并返回R[a]中的a+n）。



一旦数据被加载到一个 TOY 数组中，数组处理代码可以精确地引用数组中的第 i 个元素，如程序 6.3.4 中的指令 66 和 68 所示：将索引 i 与 a[0] 的地址相加，然后使用间接访问指令加载或存储数组元素。另一种方法是维护一个指向 a[i] 的指针，然后递增该索引以移动到下一个元素，再次使用间接访问指令访问数组元素。

下面表格中给出了两个代码示例，说明了将数组作为参数传递给函数的效用。表中的第一个例子是一个典型的数组处理程序，它的功能是在数组中寻找最大值，其中数组地址保存在 R[A] 中。它通过数组索引的迭代来逐个访问 R[A] 指向的数组中的元素，使用间接指令加载每个元素，并将其与目前为止所见的最大值进行比较，如有必要，更新该值。第二个例子是一个客户程序，从打孔的纸带读取数组，查找数组中的最大元素，并打印出最大值。构建一套支持各种数组处理的函数并不是一件困难的事情。

实际上，最初开发纸带这样的介质的动机之一是在各种应用中保存由实验测量装置产生的数据。数据处理（data processing）的想法——在打孔卡片或打孔纸带等物理介质上存储数据，然后用某种机器处理数据——早在计算机出现几十年前就一直是这样的形式。早在 20 世纪初，企业就使用打孔的卡片来存储客户数据，并于 1901 年开发出一种机器来对打孔卡片进行分类（需要一些人工干预）。的确，当今最成功的计算机公司之一 IBM（International Business Machines），在 20 世纪 50 年代推出第一台计算机之前就花了半个世纪的时间用于开发这种卡片分类器。

不难想象，即便在一个不比 TOY 更复杂的计算设备上做这些计算，也会对使用计算尺和计算器的世界产生深远影响。

939  
{  
940

用来查找最大值的函数

```
R[A] <- array address (TOY standard)
70: 7101 R[1] <- 0001
71: A90A R[9] <- M[R[A]] int i = a.length
72: 140A R[4] <- R[A]
73: 7B00 R[B] <- 0 int max = 0
74: C97C if (R[9] == 0) PC <- 7C while (i > 0)
75: 2991 R[9] <- R[9] - 1 { i = i - 1
76: 2441 R[4] <- R[4] - 1 addr of a[i]
77: AC04 R[C] <- M[R4] d = a[i]
78: 2EBC R[E] <- R[B] - R[C] if (d > max)
79: DE7B if (R[E] > 0) PC <- 7B max = d
7A: 1B0C R[B] <- R[C]
7B: C074 PC <- 74
7C: EF00 PC <- R[F] }
 return (max in R[B])
```

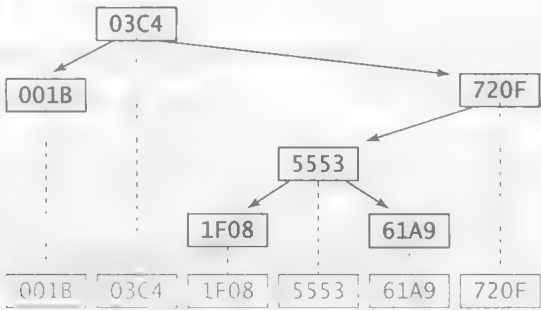
读取数组并输出  
最大值（使用程序  
5.3.5 和上面的函数）

```
5C: FF60 R[F] <- 5D; PC <- 60 read a[] from stdin
5D: FF70 R[F] <- 5E; PC <- 70 R[B] = max(a[])
5E: 9BFF R[B] to stdout write result
5F: 0000 halt
```

941

典型的数组处理代码

**链接结构** 我们学习过的其他数据结构在 TOY 中也并不是很难实现。例如，我们分析二叉搜索树（BST，见程序 4.4.3）的实现过程。为了简单起见，我们假设构建的 BST 的键值都是整数，如下图所示。

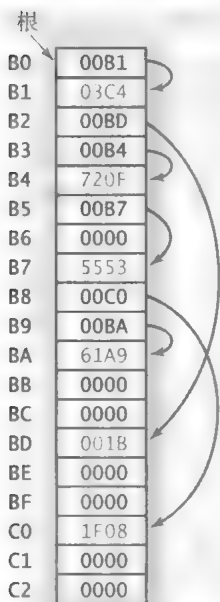


一组整数构成的一个 BST

我们假想一个客户程序，这个程序可以为输入的打孔纸带实现数据去重，也就是生成一个删除了所有重复值的新纸带。为了执行这个任务，我们每读取一个新的值，就在 BST 中搜索它是否已经存在，若没有，则将它插入 BST 中，并写在标准输出上。程序 6.3.5 给出了一个



针对这一功能的 TOY 函数实现。我们首先来分析这个函数，然后分析数据去重的客户程序。

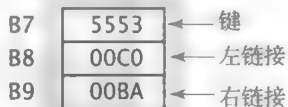


BST 表示

为了表示 BST 节点 (node)，我们使用三个 TOY 字：一个用于键值，一个用于左子树的链接，一个用于右子树的链接，0000 代表空链接。通常情况下，节点会在内存里连续保存，当然其他分布方式也是有可能的。

为了表示树，我们使用一系列在内存中连续的节点，如左图所示。每次我们需要一个新的节点时，我们将它添加到这个序列的末尾。因此，你会发现在这个图中，所有的链接都是指向下方的。

程序 6.3.5 中的 BST 搜索和插入 (search and insert) 函数有三个参数，R[A] 保存 BST 的根地址，R[B] 中保存搜索关键字，R[E] 中保存插入的新节点的地址。如果树是空的，它只是创建一个新节点，如下一段所述。如果树是非空的，则将该键与当前节点上的键进行比较 (使用间接加载指令加载该键值)，如果相等 (成功搜索)，则在 R[9] 中返回一个非零值。如果不是，则根据需要进行左链接或右链接 (再次使用间接指令来加载链接) 并循环，直到它找到一个匹配的键值或者到达 0000 链接。



BST 节点表示

#### 程序 6.3.5 链接结构：在 BST 中搜索/插入

```

80: 7101 R[1] <- 0001
81: A90A R[9] <- root x = root
82: 180A R[8] <- R[A] save link addr
83: C98F if (R[9] == 0) PC <- 8F while (x != 0) {
84: AC09 R[C] <- M[R[9]] t = x.key
85: 2CBC R[C] <- R[B] - R[C] if (t == key)
86: CC96 if (R[C] == 0) PC <- 96 return
87: 1991 R[9] <- R[9] + 1
88: 1809 R[8] <- R[9]
89: A909 R[9] <- M[R[9]] else if (t > key)
90: DC8E if (R[C] > 0) PC <- 8E x = x.left
91: 1981 R[9] <- R[8] + 1 else
92: 1809 R[8] <- R[9] x = x.right
93: A909 R[9] <- M[R[9]]
94: C083 PC <- 83
95: BE08 M[R[8]] <- R[E] }
96: BB0E M[R[E]] <- key set link to new node
97: 1EE1 R[E] <- R[E] + 1
98: 900E M[R[E]] <- 0
99: 1EE1 R[E] <- R[E] + 1
100: 900E M[R[E]] <- 0
101: 1EE1 R[E] <- R[E] + 1
102: EF00 PC <- R[F] new node(key, 0, 0)
103: return

```

调用从 FF80 开始。这个程序在以 R[A] 为根的 BST 中搜索 R[B] 给出的键值，以 R[E] 中的地址为新节点分配内存。如果搜索成功，则 R[9] 中的返回值不为零，如果搜索不成功 (并且添加了一个节点)，则返回值为零。

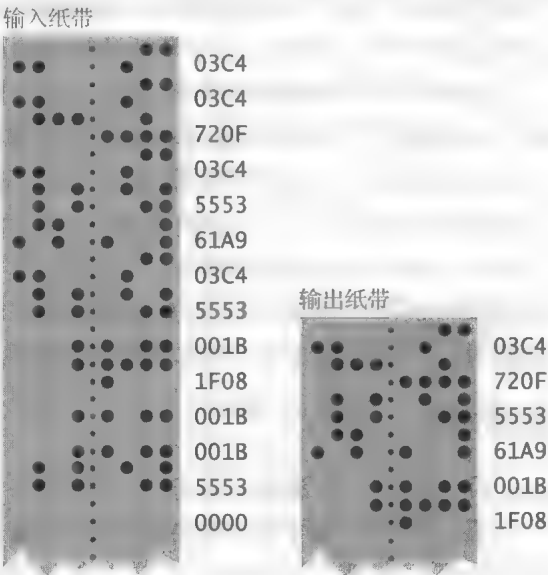
| R9   | RB   | RC   |
|------|------|------|
| 00B1 | 1F08 | 03C4 |
| 00B4 | 1F08 | 720F |
| 00B7 | 1F08 | 5553 |
| 00C0 | 1F08 | 1F08 |

PC=84 的跟踪  
搜索 1F08

一旦确定搜索键值不在树中，意味着新的节点必须在搜索终止的点处被链接到树中，则执行BST插入功能。完成此任务的技巧是保存最后一个链接的地址（指令82、88和8C处都是保存在R[8]中）。当该链接为0000时，我们可以使用指令BE08来将搜索结束的空链接替换为新节点的链接。创建一个新节点后，我们存储新键值和两个0000链接，并更新R[E]（这个代码在指令90~95中），然后返回R[9]=0。

程序6.3.5本质上是一个完整的符号表实现，在各种实际情况下它可能是快速和高效的。从许多方面来说，这个实现都比Java的版本更简单易懂。至少，这是理解链接结构本质的一个非常有用的方法。

程序6.3.5中的函数使我们的打孔纸带数据去重客户程序可以在10条TOY指令之内实现，如下所示。当你在本章开头读到TOY时，你可能不会想到它有能力进行这种有用的计算。而且，当然，这只是一个开始！



为一个打孔  
纸带去重

|     |      |                         |                           |
|-----|------|-------------------------|---------------------------|
| A0: | 7AB0 | R[A] <- B0              | root                      |
| A1: | 7EB1 | R[E] <- B1              | free space                |
| A2: | 8BFF | R[B] from stdin         | while ((b = read()) != 0) |
| A3: | CBA8 | if (R[B] == 0) PC <- A8 | {                         |
| A4: | FF80 | R[F] <- PC; PC <- 80    | x = searchInsert(b)       |
| A5: | C9A2 | if (R[9] == 0) PC <- A2 | if (x == 0) continue      |
| A6: | 9BFF | R[B] to stdout          | print(b)                  |
| A7: | COA2 | PC <- A2                | }                         |
| A8: | 0000 | halt                    |                           |

典型的符号表客户程序

为什么要学习机器语言编程？现在你已经基本了解了机器语言编程涉及的内容，在你自己开始实际编写一些机器语言程序之前，重新审视这个问题很重要。学习机器语言编程主要有以下三个基本原因：

- 机器语言程序在应用程序中仍然会经常用到。
- 在这个最低级别查看计算可以更好地揭示其本质。
- 你可以更好地理解计算机创建机器语言程序的程序，比如编译器等。

我们将在这里明确而详细地描述这些原因，并总结本章所讨论的内容。

第一，机器语言程序在关键性能中仍然会经常用到。科学家、工程师和应用程序员在不断地尝试拓展人类的计算能力，而且计算的关键部分的低级实现通常比高级语言版本要快一个或两个数量级。也许你不会参与这样的开发，但你至少应该知道可以这样做。

第二，机器语言编程通常能够捕捉到计算的本质，因为它可以剥离程序对系统和机器的依赖，直接展示出计算操作的具体步骤。机器语言程序中的数据结构要比高级语言中的更为透明。我们刚刚思考过的BST就是一个很好的例子，我们还会在练习中探索其他例子。

第三，一旦你了解机器语言，就可以考虑用Java这样的高级语言来编写可以生成程序的程序。无论何时使用计算机，都依赖于这样的程序，因为你的计算机上运行的所有内容最终都被简化为机器语言。我们将在下一节中更详细地探讨编程的这一要素。编写程序并生成程

序是一个能为你带来成就感的体验，每个人都可以尝试一下。

在本书中，我们的目的是通过描述硬件和软件之间接口的性质，来揭示计算机内部发生的事情。我们希望你把 TOY 编程视为一个了解这些重要概念的窗口，而不必应付真实世界的计算机，因为它们太过复杂。机器语言编程可以使我们可以了解很多的基本概念。或者也可以将 TOY 视作学习更复杂的计算机的必要准备。

945

本书之所以讲解机器语言编程，是因为它提供了 Java 程序和计算机之间的中间抽象层次，虽然过去几十年来计算基础设施的发展非常迅速，但是这种抽象结构没有发生变化。经过本章的学习，首先，现在你可以更好地理解编写的 Java 程序与计算机实际执行的机器语言程序之间的关系。下一节将更详细地探讨这种关系。其次，为后续的学习做好准备，你现在可以开始想象如何创建物理器件以执行这些机器语言程序。在第 7 章中，我们会设计和构建可执行机器语言程序的电路，从而彻底揭开计算机的神秘面纱。熟悉像 TOY 这样的低级机器对于理解电路如何工作起着至关重要的作用。

946

## 问答环节

问：我真的需要练习写 TOY 程序吗？

答：是的，你可能不需要像你在 Java 中编程做到的那样，但你必须理解 TOY 是另一种编程语言。当然，了解 Java 可以让你更容易学习 TOY，而且你将会看到每一门学到的语言都可以让你在学习下一个语言时更加容易。

问：我在哪里可以找到关于 TOY 的更多信息？

答：在过去的二十年里，我们的学生、助教、学科相关教师在编写本书时开发了大量的相关信息，可以在本书网站中找到。特别是如果你有兴趣，可以找到一个交互式的 TOY 模拟器，并练习用开关和灯自行输入程序。（我们用它来做教学演示。）

问：是否还有有关 TOY 的其他信息来源？

答：没有，一点也没有。因为这是一个虚构的机器。

问：我是不是应该学习一些其他的机器语言？

答：当然，你可以在真机上学习编程。正如文中所提到的，认识 TOY 是为学习真正的机器语言做更充足的准备。你可能想了解广泛使用的 IA-32 体系结构，但请注意，完整的软件开发人员手册长达数千页！

答：或者，你也可以学习 MIX，这是 D. E. Knuth 为他的经典系列书籍《计算机编程的艺术》（The Art of Computer Programming）开发的另一种假想的机器语言。你可以在这套书的前言中阅读他选择机器语言的原因；其中之一是：“一个对计算机感兴趣的人应该很好地学习使用机器语言，因为它是计算机的基础部分。”如果你学习了 MIX，在 Knuth 的书你可以读到大量只在 MIX 中表达的算法。

947

## 练习

重要提示。如果你能够有一台 TOY 机器来运行和调试程序，那么这些练习要简单得多，所以建议你先学习 6.4 节的内容，然后再回来做这些练习。

6.3.1 编写一个程序 sort3.toy，它从标准输入中读入三个整数，并按升序将其打印到标准输出。

6.3.2 编写一个程序 powers2.toy，在标准输出上打孔表示出 2 的所有正整数次幂，注意不要超过一个 TOY 字补码的表示范围。

- 6.3.3 编写一个程序 `sum_1-n.toy`，从标准输入中读入整数  $n$ ，并打印出  $1 + 2 + 3 + \dots + n$  的计算结果。
- 6.3.4 给定一个整数  $x$ ，根据克拉茨序列 (collatz sequence)，其下一个整数的定义是：如果  $x$  是偶数，则计算  $x/2$ ；如果  $x$  是奇数，则计算  $3x + 1$ ，一直循环计算，直到  $x$  为 1 (见练习 2.3.29)。编写一个程序 `collatz.toy`，它从标准输入中读取一个整数，并将它的克拉茨序列打印到标准输出。  
提示：使用右移指令进行整数的除 2 操作。
- 6.3.5 参考文中针对 `if` 和 `while` 绘制的图表，绘制一个图表以展示如何在 TOY 中实现 `for` 循环。
- 6.3.6 编写程序 `chop.toy`，从标准输入中读入一个整数  $n$ ，并将  $n$  分解为一系列 2 的幂数的和，输出这些 2 的幂数。例如，如果  $n$  是 012A，那么程序应该在纸带上打孔输出

```
0002
0008
0020
0100
```

因为  $012A = 0002 + 0008 + 0020 + 0100$ 。

- 6.3.7 编写一个程序，从标准输入中读入一个整数，求它的三次幂，然后打印出结果。对于乘法，使用 FF90 调用练习 6.3.35 中给出的乘法函数。
- 6.3.8 编写一个 TOY 代码片段，用于交换  $R[A]$  和  $R[B]$  的内容，不写入主存储器，也不能写入任何其他寄存器。提示：使用 `xor` 指令。
- 6.3.9 本题测试加载地址、加载和间接加载之间的差异。对于以下每个 TOY 程序，终止时给出  $R[1]$ 、 $R[2]$  和  $R[3]$  的内容。

|     |          |     |          |     |          |
|-----|----------|-----|----------|-----|----------|
| (a) | 10: 7211 | (b) | 10: 8211 | (c) | 10: 7211 |
|     | 11: 7110 |     | 11: 8110 |     | 11: A102 |
|     | 12: 2321 |     | 12: 2312 |     | 12: 2312 |
|     | 13: 0000 |     | 13: 0000 |     | 13: 0000 |

- 6.3.10 分析以下 TOY 程序。当停机时， $R[3]$  的值是多少？

```
10: 7101
11: 7207
12: 7301
13: 1333
14: 2221
15: D213
16: 0000
```

- 6.3.11 对于以下每个布尔表达式，给出一个 TOY 代码片段，它从标准输入中读取一个整数  $a$ ，如果对应的条件为真，则将 0001 写到标准输出；如果为 false，则将 0000 写到标准输出。

```
a = 3
a > 3
a < 3
a != 3
a >= 3
a <= 3
```

- 6.3.12 假设你将以下内容加载到 TOY 的内存中 10~17 的位置，将 PC 设置为 10，然后按下 RUN 键。

```
10: 7100 R[1] <- 0000
11: 8FFF R[F] from stdin
12: 9F15 M[15] <- R[F]
13: 82FF R[2] from stdin
14: 1112 R[1] = R[1] + R[2]
15: C016 PC <- 16
16: 91FF R[1] to stdout
17: 0000 halt
```

948

949

如果标准输入是“1112 1112”，标准输出是否会产生输出信息？如果有的话，输出是什么？

提示：第一个值存储在 M[15] 中，最终被作为代码执行。

6.3.13 将标准输入替换为 C011 C011 1112 1112，重新回答上一个问题。

6.3.14 写一个 TOY 函数，以 a、b 和 c 为参数，分别存储在 R[A]、R[B] 和 R[C] 中，计算判别式  $d = b^2 - 4ac$ ，返回的结果放在 R[d] 中。把你的代码入口放在内存中 10 的位置，用 FF90 调用练习 6.3.35 给出的乘法函数。

6.3.15 对于以下程序，分析哪些输入值能够让程序在停机前将 0001 写入标准输出。列出处于 0123 和 3210 之间的输入值。

```
10: 8AFF R[A] from stdin
11: 7101 R[1] <- 0001
12: 2BA1 R[B] <- R[A] - 1
13: 3CAB R[C] <- R[A] & R[B]
14: 9CFF R[C] to stdout
15: 0000 halt
```

答案：0200 0400 0800 1000 2000。上面这段程序对于二进制表示法中至多只有一位 1 的整数会返回 1，即十六进制表示的 0000, 0001, 0002, 0004, 0008, 0010, …, 8000。

**[950]** 6.3.16 假设你将下面的程序加载到 TOY 的内存中 10~20 位置：

```
10: 7101
11: 7A30
12: 7B08
13: 130B
14: C320
15: 1400
16: 2543
17: C51E
18: 16A4
19: A706
1A: 1717
1B: 8706
1C: 1414
1D: C016
1E: 6331
1F: C014
20: 0000
```

现在假设将 0001 0002 0003 0004 0004 0003 0002 0001 加载到内存位置 30~37，将 PC 设置为 10，然后按 RUN 键。当程序停止时，内存位置 30~37 的内容是什么？

6.3.17 填写“????”，将上一个练习中的 TOY 程序翻译成 Java 代码。

```
for (int i = n; i > ??? ; i = i ???)
 for (int j = 0; j < ??? ; j = j ???)
 a[???] = ??? ;
```

6.3.18 假设将两个向量存储在 TOY 数组中，编写一个程序计算这两个向量的点积。构建一个函数，它从 R[A] 和 R[B] 中获取数组的地址，并将点积的结果放在 R[E] 中返回。使用 FF90 调用练习 6.3.35 中给出的乘法函数。注意：你需要处理一个我们到目前还未处理过的更普遍的函数调用约束。你需要在函数调用的前后保存并恢复寄存器 R[F] 中保存的返回地址，以便在调用乘法函数时可以使用相同的寄存器作为返回地址。见练习 6.3.27。

**[951]**

6.3.19 假设将以下程序加载到 TOY 的内存位置 10~1B，并且标准输入的值为 1CAB EF00 0000 4321 1234。当将 PC 设置为 10 并按下 RUN 时，将会在标准输出上打印出什么值？

```

10: 7101 R[1] <- 0001
11: 7230 R[2] <- 0030
12: 8AFF R[A] from stdin
13: CA17 if (R[A] == 0) PC <- 17
14: BA02 M[R[2]] <- R[A]
15: 1221 R[2] <- R[2] + 1
16: C012 PC <- 12
17: 8AFF R[A] from stdin
18: 8BFF R[B] from stdin
19: FF30 see previous exercise
1A: 9CFF R[C] to stdout
1B: 0000 halt

```

6.3.20 当标准输入的值为 2CAB EF00 0000 4321 1234 时，上述练习会得到什么结果。

6.3.21 考虑下面遍历链表的代码：

```

10: 7101
11: 72D0
12: 1421
13: A302
14: 93FF
15: A204
16: D212
17: 0000

```

每个节点在内存中占据两个连续字长，一个值后跟一个链接（下一个节点的地址），链接值 0000 表示该列表的结尾。假设内存位置 D0~DB 存储了这些值：

```
0001 00D6 0000 0000 0004 0000 0002 00DA 0000 0000 0003 00D4
```

给出该程序打印出的值（将 PC 设置为 10，然后按下 RUN 键）。

答案：1 2 3 4。

952

6.3.22 指出如何通过更改上一练习中的一个内存字，以便打印“1 2 6 7”而不是“1 2 3 4 5 6 7”（链表删除）。

6.3.23 指定如何更改三个内存字（修改一个，增加两个），使练习 6.3.21 中的程序打印“1 2 3 4 8 5 6 7”（链表插入）。

6.3.24 假设 TOY 存储器保存了以下值，将程序计数器设置为 30 并按下 RUN 键。标准输出上是否会打印信息？如果有的话，信息是什么？当机器停机时，列出 R[2] 和 R[3] 的内容。

|    | 2_   | 3_   | 4_   | 5_   | 6_   |
|----|------|------|------|------|------|
| _0 | 0000 | 7101 | 7101 | 0003 | 0002 |
| _1 | 0000 | 7200 | 7200 | 0000 | 0050 |
| _2 | 0000 | 8329 | 8329 | 0005 | 0000 |
| _3 | 0000 | 1221 | A403 | 0000 | 0000 |
| _4 | 0000 | 1331 | 1224 | 0004 | 0000 |
| _5 | 0000 | A303 | 1331 | 0052 | 0000 |
| _6 | 0000 | D333 | A303 | 0000 | 0000 |
| _7 | 0000 | 92FF | D343 | 0000 | 0000 |
| _8 | 0000 | 0000 | 0000 | 0001 | 0000 |
| _9 | 005A | 0000 | 0000 | 0060 | 0000 |
| _A | 0000 | 0000 | 0000 | 0000 | 0000 |
| _B | 0000 | 0000 | 0000 | 0058 | 0000 |
| _C | 0000 | 0000 | 0000 | 0000 | 0000 |
| _D | 0000 | 0000 | 0000 | 0000 | 0000 |
| _E | 0000 | 0000 | 0000 | 0000 | 0000 |
| _F | 0000 | 0000 | 0000 | 0000 | 0000 |

6.3.25 开发一对实现下推栈的 TOY 函数,  $R[A]$  保存栈的地址,  $R[B]$  保存向栈中推入或从栈中弹出的值。

953 6.3.26 编写一个遍历 BST 的 TOY 函数, 并按照排序输出得到的键值。提示: 使用下推堆栈。

6.3.27 使用练习 6.3.25 中的解决方案来开发两个 TOY 函数: 一个将  $R[A]$  到  $R[F]$  的寄存器保存在堆栈中, 另一个从堆栈中依次恢复相应寄存器的值。使用这些函数来开发一个递归程序, 以在输出纸带上打印出类似标尺功能的线条。

6.3.28 改进我们已经实现的 BST 函数 (程序 6.3.5), 通过将每个节点的两个链接打包成一个内存字来节省空间。

954

## 创新练习

6.3.29 32 位整数。写一个 TOY 函数, 将  $R[A]$  和  $R[B]$  连接在一起看作一个 32 位二进制补码整数,  $R[C]$  和  $R[D]$  连接在一起视作第二个 32 位二进制补码整数, 将两者相加, 结果保留在  $R[A]$  和  $R[B]$  中。

6.3.30 格雷码。编写一个 TOY 程序 graycode.toy, 它从标准输入中读入一个整数  $n$  (1 到 15 之间), 然后假设有变量  $i$  从  $2^n - 1$  开始递减到 0, 打印  $(i \gg 1) \wedge i$  到标准输出。这样得到的输出序列称为  $n$  阶格雷码。参见程序 2.3.3 相关的讨论。

6.3.31 点积。计算两个数组的点积, 它们的起始位置为  $R[A]$  和  $R[B]$ , 长度为  $R[C]$ 。

6.3.32 Axy。编写一个 TOY 函数, 该函数需要输入一个标量  $a$  和一个向量  $b$ , 其中标量  $a$  保存在  $R[A]$  中, 向量  $b$  保存在一个 TOY 数组中, 数组地址保存在  $R[B]$  中, 函数返回另一个向量  $c$ , 保存  $c$  的 TOY 数组地址保存在  $R[C]$  中; 计算向量  $ab + c$ ; 并将结果留在一个 TOY 数组中, 并在  $R[D]$  中保存这个数组的地址。

6.3.33 一次性密码本。在 TOY 中实现一次性密码本的功能, 用以加密和解密 256 位消息。假定密钥被存储在存储器位置 30~3F 中, 输入数据由 16 个 16 位整数组成。

6.3.34 找到不同的数。假设在标准输入上输入了  $2n + 1$  个 16 位整数的序列, 其中  $n$  个整数出现且仅出现两次, 仅有一个整数只出现一次。编写一个 TOY 程序来找出这个数。提示: 将所有的整数进行异或。

6.3.35 有效的乘法。实现你在小学时学过的算法, 把两个整数相乘。具体来说, 让  $b_i$  表示  $b$  的第  $i$  位, 这样:

$$b = (b_{15} \times 2^{15}) + (b_{14} \times 2^{14}) + \cdots + (b_1 \times 2^1) + (b_0 \times 2^0)$$

现在, 要计算  $a \times b$ , 使用分配定律:

$$a \times b = (a \times b_{15} \times 2^{15}) + (a \times b_{14} \times 2^{14}) + \cdots + (a \times b_1 \times 2^1) + (a \times b_0 \times 2^0)$$

955

你很自然地会想, 这似乎将一次乘法操作变成了 32 次乘法操作, 对于 16 位中的每一位做两次。幸运的是, 这 32 个乘法中的每一个都是一个非常特殊的类型, 因为  $a \times 2^i$  与  $a$  左移  $i$  位相同。由于  $b_i$  是 0 或 1, 因此第  $i$  项是  $a \ll i$  或者是 0。

答案:

```

90: 7101 R[1] <- 0001
91: 7C00 R[C] <- 0000 c = 0
92: 7210 R[2] <- 0010 for (i = 16; i > 0; i--)
93: 2221 R[2] <- R[2] - 1 {
94: 53A2 R[3] <- R[A] << R[2] t = a * 2^i
95: 6482 R[4] <- R[B] >> R[2]
96: 3441 R[4] <- R[4] & 1

```



```

97: C41B if (R[4] == 0) PC <- 99 if b[i] == 1
98: 1CC3 R[C] <- R[C] + R[3] c += t
99: D293 if (R[2] == 0) PC <- 93 }
9A: FF00 return

```

6.3.36 幂函数。使用上一个练习的乘法函数，实现一个计算  $a^b$  的 TOY 函数，参数  $a$  和  $b$  的值分别存在  $R[A]$  和  $R[B]$  中。

6.3.37 多项式求值。使用前面两个练习的幂函数和乘法函数，实现一个 TOY 函数，该 TOY 函数使用一个 TOY 数组保存系数  $a_0, a_1, a_2, \dots, a_n$ ，用一个整数表示自变量  $x$ ，其中数组的地址保存在  $R[A]$  中， $x$  的值保存在  $R[B]$  中。计算以下多项式：

$$p(x) = a_n x^n + \dots + a_2 x^2 + a_1 x^1 + a_0 x^0$$

将结果保存在  $R[C]$  中并返回。多项式求值是早期计算机（准备弹道表）解决的主要问题之一。

6.3.38 霍纳方法是解决多项式求值问题时比直接求值更高效、更容易编码的一种巧妙的方法。这个方法的基本思想是合理地排列乘法的组合方式，如下例所示：

$$p_4 x^4 + p_3 x^3 + p_2 x^2 + p_1 x^1 + p_0 x^0 = (((p_4)x + p_3)x + p_2)x + p_1)x + p_0$$

基于这个想法实现一个 TOY 函数，只使用  $n$  次乘法来为一个  $n$  阶多项式求值。这种方法是在 19 世纪由英国数学家威廉·霍纳（William Horner）发表的，但在此之前的一个多世纪艾萨克·牛顿（Isaac Newton）就在使用它了（见练习 2.1.31）。 956

6.3.39 号码转换。实现一个使用霍纳方法的 TOY 程序（参见程序 6.1.1），它将十进制整数转换为二进制表示。从标准输入中以十六进制的形式（格式为 000x）读取一个十进制整数，然后在标准输出上打印出它的二进制编码，一次打印一位。 957

## 6.4 TOY 虚拟机

考虑到 TOY 是一个虚拟机，我们如何能够运行和调试程序呢？在本节中，我们将为这个问题提供一个完整的答案，然后讨论其意义。在本节中你将看到，我们可以轻松地编写一个称为虚拟机的 Java 程序，它可以运行任何 TOY 程序。事实上，正如你所看到的，虚拟机定义了 TOY 的实现方式，它精确地描述了每个 TOY 指令的效果，并且始终保存着 TOY 机器的完整的状态信息。

虚拟机定义的是这样的一种机器：它像真机一样执行程序，但不需要与任何物理硬件直接对应。通常我们使用这个术语来宽泛地指代机器以及实现它的软件（或硬件）。

虚拟机的概念在一开始或许会令人不太适应，为确保读者能够完全理解这一概念，我们会完整地讲述这一概念的所有细节，而虚拟机的概念也是第 5 章中讨论的计算理论基本思想的核心。事实上，邱奇-图灵理论意味着任何计算系统都可以执行相同的计算，只要内存足够大，因此每台计算机都可以视作其他计算机的虚拟机。

虚拟机的核心概念是处理程序的程序。你可能还记得，在停机问题的证明中，我们引入了一个想法，即一个程序可以将另一个程序作为输入。这个想法起初似乎有点奇怪，实际上，这个想法是一个基本的计算机科学概念，我们将在本节中详细探讨。

作为热身，我们讨论一些简单的 TOY 编程概念的实际应用，其中包括一个重要的和难以避免的常见问题。我们会讨论这些问题与 Java 存在很多相似与关联，进而开始讨论本节的核心部分 TOY 虚拟机。最后我们讨论这一模型对现代和未来计算机的影响。

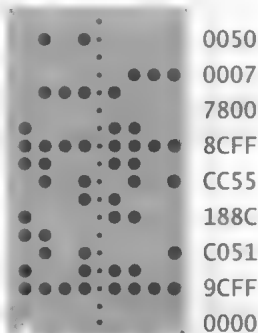
本节涵盖了很多方面的内容，从 TOY 到停机问题，再到服务器农场和云计算，接下来我们将会揭示这些概念和术语背后的深层含义，并诠释它的能力、简洁以及重要性。篇

958

幅所限，我们无法对它们进行详尽的解释，但对于从事计算的人来说，理解这些术语非常重要。

**启动和导出** 首先我们来研究所有类似 TOY 的冯·诺依曼机器都具有的本质特征——它可以处理任何类型的数据，而不仅仅是数字。事实上，一个程序的指令可以是另一个程序的数据。具体来说，我们来分析 TOY 编程环境中这个属性的两个实际结果。

**启动。**思考右图所示的打孔纸带，看起来很熟悉吗？在运行时，可能会将这些 4 位十六进制数字中的一部分识别为 TOY 指令。实际上，我们之前已经看到过它们——它们是程序 6.3.3（计算输入纸带上数字的总和）的指令序列，在最前面的是第一个指令的地址和指令的数量。但是，正如我们从本章开始就强调的那样，计算机内给定二进制序列的含义取决于上下文。因此，程序也可以将这些数据解读为二进制补码。事实上，这个纸带也可以作为程序 6.3.3 的输入来计算其总和：



|   |      |
|---|------|
| • | 0050 |
| • | 0007 |
| • | 7800 |
| • | 8CFF |
| • | CC55 |
| • | 188C |
| • | C051 |
| • | 9CFF |
| • | 0000 |

输入数据 (?)

$$50_{16} + 7_{16} + 7800_{16} + (-7301_{16}) + (-33AB_{16}) + 18CC_{16} + (-3FAF_{16}) + (-6301_{16})$$

更重要的是，这个纸带也可以作为读取数组程序（程序 6.3.4）的输入——我们认为纸带是一个程序，并把程序 6.3.4 当作一个将其他程序加载到内存的程序。事实上，我们可以为我们设计过的每个 TOY 程序制作纸带，并以这种方式将它们加载到内存中。

又或者，我们可以为 10~FF 之间的所有内存位置设置一个纸带，并用我们的代码和数据填充整个内存。这个过程被称为启动计算机——这个术语一直使用到现在。我们使用某个特殊的方法（在 TOY 中是开关）将一段小程序加载到内存中，然后它可以从外部设备加载内存的其余部分。

在下面“启动和导出”的表格的左边是从程序 6.3.4 获得的代码，我们对它做了一些简单修改，从一个函数变为一段启动代码（boot code）。它会在计算机加电的最开始，由程序员通过开关输入到计算机中，并成为计算机运行的第一段代码。在计算机的发展历程中，典型的做法是预留内存的最前面几个字（在我们的例子中为 00~0F）用作启动程序。像 TOY 这样的计算机通常都会在前面的控制板上写上一段这样的程序，因为这将是唯一需要通过开关进入的程序。程序员需要通过拨动开关加载这段程序，他们让手指像钢琴演奏家一样（好吧，不完全是）在开关上滑动，并以这样的输入速度引以为傲。

959

除了不作为函数打包之外，启动程序与程序 6.3.4 有两个不同之处。第一个区别是没有必要存储长度（长度信息是实现数组操作才需要的）。第二个（更深刻的）区别在于启动程序以语句 EA00 结束。由于 R[A] 包含加载的第一个字长的地址，该指令用于将控制权交给刚加载的数据，这是冯·诺依曼机器中一个典型的将数据改变成指令的例子。

现代计算机使用基本相同的启动过程，所以这个术语直到今天仍在使用。当你重新启动（reboot）手机或平板电脑或计算机时，操作系统和许多基本应用程序将通过一个小程序从处理器外部的存储器加载到设备中，该程序通过一些特殊的过程将其自身加载到处理器内存中。然后通过分支指令将控制权交给该程序。

**导出。**如果要编写一个程序，用于生成打孔的纸带以供启动程序加载，那么简单的方法就是修改启动程序代码，让它在纸带上打出加载地址和程序长度，并循环打出待加载程序的每个内存字（原来的启动程序是读取这些信息）。这些更改在下图右下方显示的导出程序中

以**粗体**突出显示。前两个字是指令加载地址和长度——程序员可以将它们（通过开关）设置为任何值。值 10（地址）和 EF（长度）指定了一个“完全导出”（full dump）：我们只将从 10 到 FE 的内存字导出，因为 00~0F 是为导出 / 启动程序本身预留的空间，而 FF 预留给标准输入和标准输出。这个过程被称为导出（dumping）内存的内容。

|                                          |                         |                                          |
|------------------------------------------|-------------------------|------------------------------------------|
| 02: 8AFF R[A] <i>from stdin</i>          | <i>read/write</i>       | 00: 7A10 R[A] <- 0010                    |
| 03: 8BFF R[B] <i>from stdin</i>          | <i>address of a[]</i>   | 01: 7BEF R[B] <- 239 <sub>10</sub>       |
| 04: 7101 R[1] <- 0001                    | <i>b=length of a[]</i>  | 02: 9AFF R[A] <i>to stdout</i>           |
| 05: 7900 R[9] <- 0000                    | <i>i = 0;</i>           | 03: 9BFF R[B] <i>to stdout</i>           |
| 06: 22B9 R[2] <- R[B] - R[9]             | <i>while (i &lt; b)</i> | 04: 7101 R[1] <- 0001                    |
| 07: C20D <i>if (R[2]==0) PC &lt;- 0D</i> | <i>{</i>                | 05: 7900 R[9] <- 0000                    |
| 08: 1CA9 R[C] <- R[A] + R[9]             | <i>address of a[i]</i>  | 06: 22B9 R[2] <- R[B] - R[9]             |
| 09: 8DFF R[D] <i>from stdin</i>          | <i>read/write a[i]</i>  | 07: C20D <i>if (R[2]==0) PC &lt;- 0D</i> |
| 0A: BD0C M[R[C]] <- R[D]                 | <i>i = i + 1</i>        | 08: 1CA9 R[C] <- R[A] + R[9]             |
| 0B: 1991 R[9] <- R[9] + 1                | <i>}</i>                | 09: AD0C R[D] <- M[R[C]]                 |
| 0C: C006 PC <- 06                        |                         | 0A: 9DFF R[D] <i>to stdout</i>           |
| 0D: EA00 PC <- R[A]                      |                         | 0B: 1441 R[9] <- R[9] + 1                |
|                                          |                         | 0C: C006 PC <- 06                        |
|                                          |                         | 0D: 0000 <i>halt</i>                     |

启动（见程序5.3.5）

启动和导出

导出

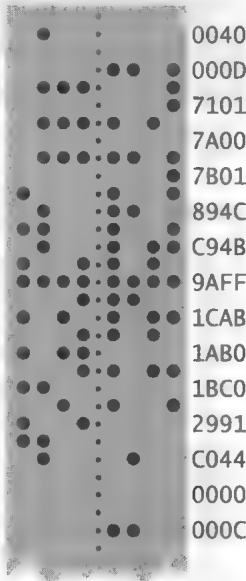
对于 TOY 这类计算机，这两个简单的过程大大简化了程序员的工作流程。编程工作首先需要通过使用开关输入引导程序，然后运行引导程序以从纸带加载各种程序。如果某一天的工作用到了新的代码（通过开关输入），那么可以对程序做一些改动（在我们的例子中，需要修改内存位置 00~03 和 09~0A）将启动转换为导出，将新的代码打孔成纸带来保存，以供其他时间使用。

例如，在输入和调试程序来计算斐波那契数列（程序 6.3.2）之后，我们可以保存这个程序。需要注意的是，该程序包含的 0D 个字存储在 40 到 4C 位置。使用开关将 00 的指令改为 7A40，01 的指令改为 7B0D，然后将地址开关设置为 00，最终按下 RUN 键以运行导出程序，生成如右图的纸带，随后，我们可以随时从这个纸带启动以加载这个程序。

不难想象程序员很快就开发出了一系列包含各类代码的打孔纸带，你可以将这样的集合看作是外部存储（external storage）的早期形式，而引导程序是安装程序（installer）的一个早期形式，你现在经常使用这些技术来将程序放到移动设备上。

**注意** 接下来，我们讨论一个例子来说明虽然冯·诺依曼架构在许多方面令人惊叹，但也可能存在危险。

像 TOY 机上这样的典型工作流程可能是让科学家开发一个处理实验数据的程序，然后在数周或数月的时间内使用该程序来实际处理数据。由于这个活动只需要运行加载程序（通过开关输入启动程序，或者只是检查发现它已经被加载，然后就直接使用它），然后将数据加载到纸带阅读器上以供程序处理，所以经常是雇佣机器操作员（不需要有编程基础或科学技能）来做这些工作。机器操作员可以在机器上加载一个科学家的程序，然后在不同的实验数据集上运行它们，这种工作可能需要持续几个小时。



程序 6.3.2 的导出

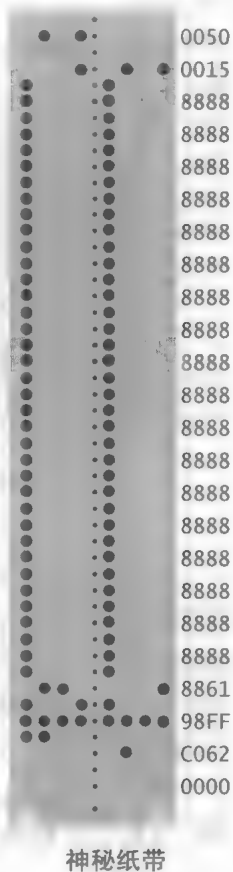
现在想象一个人需要运行一个大型程序，程序的开始部分使用程序 6.3.4（驻留在内

存位置 60 到 6D) 来加载一个数组。有一天,一位同事要求使用该程序,并向操作员提供右图所示的纸带。操作员将纸带放在阅读器上,按下 RUN 之后(也许)去吃午餐。纸带的内容是从 50 开始加载一个 21 个字的数组。会发生什么?

如果我们采用计算机的观点,分析这种情况并不困难。我们都知道,无论 PC 指定地址的指令是什么,机器都会抓取指令、递增 PC、执行指令,然后再继续重复这个周期,直到遇到停机指令。在这个例子中,引导程序会读到预期的地址和长度,然后加载纸带上的 16 个字,但是当 R[9] 是 5F 然后递增时,我们需要仔细分析这个情况。

当我们递增 5F 时,我们得到结果 60,所以数据的下一个字存储在 60 中,然后再下一个字存储在 61 中,以此类推,如下表格所示。这个过程产生的结果可能是意料之外的,因为读取数组的程序正在覆盖自己。最终,程序会开始执行这些指令,而这些指令原先是纸带上的数据。在这个例子中,指令会跳转回到先前的指令(这个指令原先也是纸带上的数据),运行的结果便是机器进入无限循环,以尽可能快的速度在纸带上打孔 8888。这个操作员吃完午餐后回来,一定想不到会看到这样的情况。

这个虚构的故事简单地展现了控制计算机中出现的意外。打孔 8888 的三行循环可能是任何程序,而且这个过程也不必一定要在启动程序中出现。针对我们的数组输入程序或任何其他的读取和保存数据的程序,你都可以按照类似的思路来控制程序的执行逻辑。



962

指令66~68的效果

| R[C] →   | 60   | 61   | 62   | 63   | 64   |                                                     |
|----------|------|------|------|------|------|-----------------------------------------------------|
| 60: 8AFF | 8888 | 8888 | 8888 | 8888 | 8888 | data<br>R[8] <- M[61]<br>R[8] to stdout<br>PC <- 62 |
| 61: 8BFF | 88FF | 8888 | 8888 | 8888 | 8888 |                                                     |
| 62: 7101 | 7101 | 7101 | 8861 | 8861 | 8861 |                                                     |
| 63: 7900 | 7900 | 7900 | 7900 | 98FF | 98FF |                                                     |
| 64: 22B9 | 22B9 | 22B9 | 22B9 | 22B9 | C062 |                                                     |
| 65: C268 | C268 | C268 | C268 | C268 | C268 |                                                     |
| 66: 1CA9 | 1CA9 | 1CA9 | 1CA9 | 1CA9 | 1CA9 |                                                     |
| 67: 8DFF | 8DFF | 8DFF | 8DFF | 8DFF | 8DFF |                                                     |
| 68: BD0C | BD0C | BD0C | BD0C | BD0C | BD0C |                                                     |
| 69: 1991 | 1991 | 1991 | 9901 | 1991 | 1991 |                                                     |
| 6A: C064 | C064 | C064 | C064 | C064 | C064 |                                                     |
| 6B: 1AA9 | 1AA9 | 1AA9 | 1AA9 | 1AA9 | 1AA9 |                                                     |
| 6C: BB0A | BB0A | BB0A | BB0A | BB0A | BB0A |                                                     |
| 6D: EF00 | EF00 | EF00 | EF00 | EF00 | EF00 |                                                     |

通过缓冲区溢出攻击控制 TOY 机器

几十年来困扰计算机用户的计算机病毒与这种情况类似。在很多情况下,计算机系统很容易被攻击,将控制转移到内存中某个被设定为数据的区域,就会产生各种可怕的后果。举一个例子,对于 C 语言编写的典型程序,在用户提供的字符串参数比函数期望的要长时,就会受到缓冲溢出攻击(buffer overflow attack)。由于函数代码出现在原本用来保存字符串的内存缓冲区之后,恶意用户可以使用比预期更长的字符串对程序进行编码,就像我们的例子一样。系统将控制转移到函数应该在的存储器位置,但这其实就是将控制交给了恶意用户。如果此时的代码是病毒(virus)程序,则该代码会试图连接并感染其他计算机,导致

情况迅速升级。记录在案的这类案件层出不穷，困扰了数百万计算机用户，并且问题仍在继续。

我们不能写一个程序来检查这种情况吗？不是有病毒防护软件的帮助吗？遗憾的是，这样的软件只是扫描已知的病毒，而且也不能确定给定的指令序列可能会做什么。的确，一般来说，正如停机问题的不确定性证明一样（参见 5.4 节），不可能编写一个程序来检查某个程序是否是病毒。

963

**处理程序的程序** 好消息是，在许多情况下，编写将其他程序作为输入（或输出）的程序是非常有用的。自从在 1.1 节讨论 Java 编译器和 Java 虚拟机以来，我们已经非正式地讨论了这些程序，现在结合 TOY 的知识，我们可以提供更多的细节。

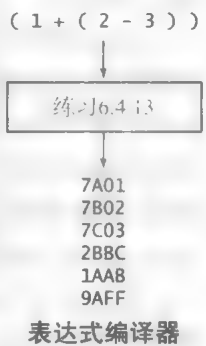
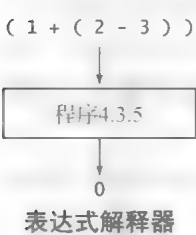
**汇编程序。**用十六进制数字直接编程很不方便且易产生错误，所以许多计算机的初期开发使用的是汇编语言（assembly language），它使得程序员可以为操作指令和机器地址建立符号名称。汇编程序是将汇编语言程序作为输入并生成机器语言程序作为输出的程序。在 Java 中编写 TOY 的汇编程序并不困难（参见练习 6.4.13）。虽然在 TOY 代码中做这项工作有点难度，但是早期的程序员在面对各种新设计的计算机时都要应对这个困难。汇编语言程序设计在今天仍然是普遍存在的。

**解释器。**解释器（interpreter）是用来直接执行用编程语言编写的代码的程序。我们已经看到了一个解释器的简单例子：程序 4.3.5，它的功能是用来计算算术表达式。一个算术表达式用一种简单的编程语言来指定一个计算任务，程序 4.3.5 执行这个计算任务。这个过程非常简单，你可以想象在 TOY 中如何实现它（参见练习 6.4.15）。许多现代编程语言程序都使用解释器来处理。使用基于解释器的系统的主要原因是它可以是交互式的——你可以一次输入一个指令。不使用这种系统的一个主要原因是它可能是低效的，因为每一条遇到的源代码语句都必须被解析和处理。

**编译器。**编译器（compiler）是将一种计算机语言（通常是源代码）转换成另一种计算机语言（通常是机器语言）的程序，通常用于创建可执行程序。为了更好地理解这个概念，我们鼓励你完成练习 6.4.14，在这里你需要把算术表达式求值器转换成编译器。例如，当解释器在遇到“+”号时执行加法，编译器将输出执行加法的机器指令。在整个源程序被处理后，我们得到的结果是一个机器语言程序。大多数在产业界中使用的编程系统都是基于编译的，因为现代编译器可以生成与手写代码解决方案一样高效（甚至更高效）的机器语言程序。

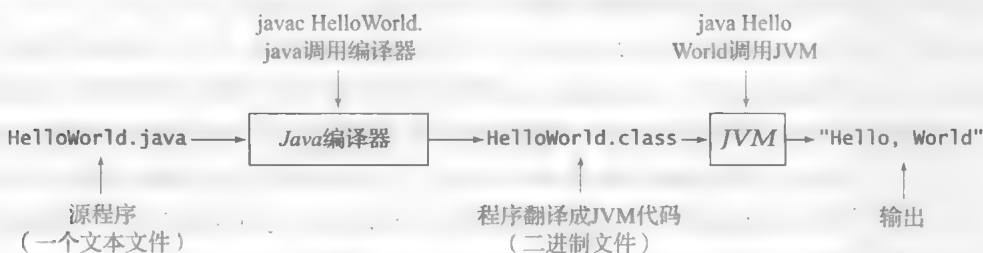
**虚拟机。**虚拟机定义的是这样的一台机器：它可以像真机一样执行程序，但不需要与任何物理硬件直接对应。当然，TOY 是一个虚拟机！从历史上看，这个术语已经发生了很大变化。因此，为了不重写已有的软件，每一个新的计算机设计都配有一个名为仿真器（emulator）的软件或硬件，其可以运行为其前身编写的程序。虚拟机最早的用途之一是分时操作（timesharing），这种软件会给出计算机多个副本的错觉，而实际上所有这些都运行在一台计算机上。虚拟机在早期的另一个用法是，用作高级语言和机器硬件之间的抽象中间层（这是我们接下来要研究的 Java 方法）。在现代计算中，术语虚拟机（virtual machine）是涵盖了所有这些含义的。

**JVM。**Java 虚拟机是一个典型案例。Java 系统的设计者知道，与其为每一种处理器开



964

发一个 Java 编译器，不如定义一个具有许多真实机器特性的虚拟机（寄存器、存储器、程序计数器、执行算术和逻辑操作的指令；在寄存器和存储器之间传递信息；并执行分支和跳转），在这个虚拟机上定义一个被称为字节码（bytecode）的指令集，并由解释器有效地执行字节码程序。然后他们可以把所有的精力都放在开发一个从 Java 到 JVM 的编译器上。以下图表描述了他们遵循的过程。为了使 Java 在任何特定的计算机上工作，为 JVM 编写一个解释器就足够了——这比为 Java 开发一个新的编译器要容易得多。尽管 Java 是几十年前开发的，但即使在今天，它也能成功地用于新机器上。作为该思想的实用性的进一步证据，程序员们甚至开发了编译到 JVM 的新语言。这些语言可以运行在任何可以运行 Java 程序的设备上。



处理程序的程序

这类程序的形式是多种多样的，而且都是非常有趣的，你从事的计算任务越多，遇到它们的情况就越多。程序和数据之间的唯一区别就是上下文。在 TOY 中，如果 PC 中保存了某个内存字的地址，则这个字就是一条指令；否则，它就是数据。冯·诺依曼机器的这个本质特征是现代计算的一个关键属性。图灵构思了这个想法，冯·诺依曼在实践中把握了它的重要性，而整个世界从中得利至今。

在目前的情况下，我们最重要的兴趣在于 Java 和 TOY 之间的关系，所以我们接下来讨论这个话题。

**Java 中的 TOY** 你需要仔细研究 Java 程序 6.4.1，从而获得一个直观的感受，这个程序就是 TOY 机器的实现，我们用它来实现和调试本书中的所有 TOY 程序（我们鼓励你自己使用它来实现和调试一些 TOY 程序）。你会惊讶于这个程序怎么这么容易理解。事实上，我们设计 TOY 时主要考虑因素之一就是要足够简单，尽量用一页纸就能够描述清楚。除了标准输入和标准输出的代码之外，这个程序已经是完整的了。这里我们省略输入输出代码，以便我们专注于虚拟机的核心部分。

**解析 TOY 指令。**假设我们在一个整型变量 IR 中保存一个 TOY 指令。它是一个 32 位的值，但 TOY 指令只是 16 位，所以只用处理最右边的 16 位。通过我们在 6.1 节中所学习的移位和掩码操作，我们可以分离出操作码、寄存器和地址，以备以后使用：

```
int op = (IR >> 12) & 0xF;
int d = (IR >> 8) & 0xF;
int s = (IR >> 4) & 0xF;
int t = (IR >> 0) & 0xF;
int addr = (IR >> 0) & 0xFF;
```

对于任何特定的指令，我们可能会使用 s 和 t 或 addr，但不能同时使用，但最简单的方法就是对每条指令都把它们的三个值计算出来。

**机器的状态。**我们从一开始就注意到 TOY 机器的行为完全由寄存器（特别是 PC）的内容和存储器的内容决定。这个事实自然引起我们在程序 6.4.1 中选择实例变量：我们使用一个包含 16 个整型值的数组作为寄存器，一个包含 256 个整型值的数组作为内存，一个单独的整



型值用于 PC。再次，我们实际上只使用这些值的最右边的 16 位（参见本节结尾的问答环节）。

需要注意的是，当我们考虑到标准输入时，即使机器本身是一个有限状态的机器，机器状态的大小也是不受限的。内存和寄存器只有 4160 位，但输入纸带上的位数是无限的。

程序 6.4.1 TOY 虚拟机（无标准输入输出）

```
public class TOY
{
 private int[] R = new int[16];
 private int[] M = new int[256];
 private int PC;

 public TOY(String filename) // 构造函数; 见文本
 public void run()
 {
 while (true)
 {
 int IR = M[PC]; // 提取
 PC = (PC + 1) & 0xFF; // 增量

 int op = (IR >> 12) & 0xF;
 int d = (IR >> 8) & 0xFF;
 int s = (IR >> 4) & 0xFF;
 int t = (IR >> 0) & 0xFF;
 int addr = (IR >> 0) & 0xFF;

 if (op == 0) break;
 switch (op)
 {
 case 1: R[d] = R[s] + R[t]; break;
 case 2: R[d] = R[s] - R[t]; break;
 case 3: R[d] = R[s] & R[t]; break;
 case 4: R[d] = R[s] ^ R[t]; break;
 case 5: R[d] = R[s] << R[t]; break;
 case 6: R[d] = (short) R[s] >> R[t]; break;
 case 7: R[d] = addr; break;
 case 8: R[d] = M[addr]; break;
 case 9: M[addr] = R[d]; break;
 case 10: R[d] = M[R[t] & 0xFF]; break;
 case 11: M[R[t] & 0xFF] = R[d]; break;
 case 12: if ((short) R[d] == 0) PC = addr; break;
 case 13: if ((short) R[d] > 0) PC = addr; break;
 case 14: PC = R[d] & 0xFF; break;
 case 15: R[d] = PC; PC = addr; break;
 }

 R[d] = R[d] & 0xFFFF;
 R[0] = 0;
 }
 }

 public static void main(String[] args)
 {
 /* 见练习 6.4.2 */
 }
}
```

|      |         |
|------|---------|
| R[]  | 寄存器     |
| M[]  | 主存      |
| PC   | 程序计数器   |
| op   | 操作码     |
| d    | 结果寄存器   |
| s    | arg 寄存器 |
| t    | arg 寄存器 |
| addr | 地址字段    |

启动机器。为了在运行 TOY 模拟器时略微简化我们的工作流，我们在构造函数中启动机器，如下面的代码所示。客户程序提供文件名和 PC 的初始值作为参数——文件中存储指令序列，每行包含内存地址和相应的指令，之间用冒号和空格分隔。右边的例子程序 fib.toy 是一个 13 字节的程序——与之前的程序 6.3.3 相同，它需要加载到 M[40—4C] 中，并通过将 PC 设置为 40 来运行。也就是说，我们把 TOY 程序保存在一个文件中，然后通过构造函数中提供该文件的文件名和 PC 的初始值以实现从该文件启动。构造函数通过在指定的内存位置保存指令序列来加载 TOY 内存。构造函数完成后，TOY 程序就可以通过调用 run() 方法来执行了。

```
% more fib.toy
40: 7101
41: 7A00
42: 7B01
43: 894C
44: C94B
45: 9AFF
46: 1CAB
47: 1AB0
48: 1BC0
49: 2991
4A: C044
4B: 0000
4C: 000C
```



```

public TOY(String filename, int pc)
{
 PC = pc & 0xFF;
 In in = new In(filename);
 while (in.hasNextLine())
 {
 String line = in.readLine();
 String[] fields = line.split("[:\\s]+");
 int addr = Integer.parseInt(fields[0], 16) & 0xFF;
 int inst = Integer.parseInt(fields[1], 16) & 0xFFFF;
 M[addr] = inst;
 }
}

```

967  
}  
968

### TOY 虚拟机的构造函数

这种特殊的启动过程实际上反映了最终出现在真实计算机上的方法，不同之处是真实计算机会使用一些输入设备来启动，而启动的过程因设备不同而不同。详细机制在此不再赘述，我们的主要兴趣在于当内存被加载并且 PC 已经用指定的地址初始化时会发生什么。在我们的例子中，我们调用 `run()` 方法。这个动作模拟一个操作员在输入或引导程序并将开关设置为开始地址之后按下 RUN 按钮的动作。

运行 `run()` 方法是模拟器的核心，其实现非常简单。我们把 PC 中地址的指令取到一个整型变量 IR 中，然后递增 PC（在一条语句中完成这两个操作）。然后我们解码指令的所有组成部分（操作码、结果寄存器、参数寄存器和地址），如刚刚所述。根据提取的这些信息来改变机器状态，这个过程是在 `switch` 语句中实现的，也就是说，我们在这里模拟指令的执行。很容易看出每条指令的作用，因为每条指令的 Java 代码与我们在第一次引入时的描述是一致的。

接下来发生的事情完全取决于指令及其引起的机器状态的变化。与 TOY 按照其 PC 执行指令的方式相同，虚拟机根据其 PC 变量执行指令，直到遇到停机指令（操作码 0）。对于我们的示例程序 `fib.toy`，如右图所示，其结果正如预想的一样，会在标准输出上打印出斐波那契数。

```

% java TOY fib.toy
0000
0001
0001
0002
0003
0005
0008
000D
0015
0022
0037
0059

```

```

private void stdin(int addr, int op, int t)
{
 if ((addr == 0xFF && op == 8) || (R[t] == 0xFF && op == 10))
 M[0xFF] = Integer.parseInt(StdIn.readString(), 16) & 0xFFFF;
}
private void stdout(int addr, int op, int t)
{
 if ((addr == 0xFF && op == 9) || (R[t] == 0xFF && op == 11))
 StdOut.printf("%04X\n", M[0xFF]);
}

```

969

### TOY 虚拟机的标准输入和标准输出

标准输入与输出。启动过程中加载的程序来自于文件，所以我们可以使用 `StdIn` 作为标准输入，`StdOut` 作为标准输出。具体而言，我们需要：在一个加载指令（操作码 8）或间接加载指令（操作码 A）访问内存位置 FF 时从标准输入读取一个值；并在存储指令（操作码 9）或间接存储指令（操作码 B）访问内存位置 FF 时向标准输出写入一个值。相应的代码实现如上 `stdin()` 和 `stdout()` 方法所示。要将标准输入和标准输出添加到程序 6.4.1 中，只需在主语句 `switch` 之前添加调用 `stdin(addr, op, t)`，之后添加调用 `stdout(addr, op, t)`。另一种

实现方法是，我们可以在访问内存之前检查目标地址并执行相应的操作，我们将在第 7 章中看到，实际上硬件更倾向于后面这种实现方案。实现的细节不再赘述——我们这里只是为了准确地模拟机器的行为。还需要说明的是，为了避免不必要的工作量，我们并没有做类似打孔纸带那样的二进制输入和输出（参见练习 6.4.3）。

左边的例子展示了 `sum.toy` 的内容，这个文件是一个 7 字节的程序，实现细节如程序 6.3.3 所示。`sum.txt` 是这个程序的采样数据，将在标准输入中出现，模拟一条打好孔的纸带。当需要执行这段代码时，`TOY.java` 中的测试代码将程序加载到 `M[50—56]`，将 `PC` 设置为 50，并调用 `run()`。模拟标准输入很简单——我们重定向标准输入来自 `sum.txt`，如右侧的代码所示。每次执行 `8CFF` 指令时，模拟器会调用 `stdin()` 从标准输入获取数据并填入 `M[FF]`，所以程序实现的功能就是读取所有数字并将它们相加求和。最后，用 `98FF` 指令将结果写入标准输出并停机。你可以

```
% more sum.toy
```

```
50: 7800
51: 8CFF
52: CC55
53: 188C
54: C051
55: 98FF
56: 0000
```

```
% more sum.txt
```

```
0001
0008
001B
0040
007D
00D8
0157
0200
0000
```

看到，对于任何 TOY 程序，大抵都是这样的执行过程。

```
% java TOY sum.toy 50 < sum.txt
0510
```

开发 TOY 程序。如果需要的话，我们可以轻

松地对程序 6.4.1 进行修改，以便获取程序运行过程中 `PC` 的路径、寄存器内容和受影响的内存单元等，也可以在任何需要的时候提供内存导出（参见练习 6.4.3）。在 20 世纪 80 年代初，程序员花费了大量的时间去处理内存导出，因为这是弄清机器内部发生的各种错误的唯一方法。事实上，你可能会将程序 6.4.1 扩展，把它当成 TOY 的开发环境，可以使用它来开发和调试 TOY 程序。此外，你可以为它添加代码以获得任何需要的信息，用来分析你的程序正在做什么。所有这一切都比在真实的 TOY 机器上（在这个例子中，这是不可能的，因为没有真正的 TOY 机器）做要容易得多。你可以在程序 6.4.1 的基础上继续添加代码，只要是你认为在程序开发的过程中可能会需要的功能都可以试着实现。事实上，你可以在本书网站上找到一个模拟器（它也是由学生编写的，与你一样的初学者），它包含了一个显示开关和指示灯的图表；支持跟踪和内存导出，一次一个指令地执行程序；并提供了许多其他功能。

[970]

摩尔定律。在过去的六七十年里，最先进的计算机的速度和内存大概每 18 个月翻一番。这个经验法则叫作摩尔定律。它给我们带来了一个持续的挑战：我们如何构建一台新的计算机，同时又不会浪费我们在开发软件方面的所有努力？虚拟机在这个过程中起着至关重要的作用，因为任何新计算机的设计早期就是在旧计算机上构建一个类似于程序 6.4.1 的虚拟机。这种方法有几个好处：

- 可以在新计算机创建之前开发在其上运行的软件。
- 计算机一旦建成，可以根据虚拟机的操作来检查其实际操作，以验证硬件的正确性。
- 为新计算机开发的早期软件很可能是旧计算要的虚拟机！这样，为旧计算机开发的任何软件都可以在新计算机上运行。

例如，我们可以创建一个 TOY 程序来实现 TOY 虚拟机本身吗？当然！尽管程序 6.4.1 是一个相对简单的 Java 程序，但翻译其他 Java 程序也并不困难（见练习 6.4.10）。既然我们拥有了一个 TOY 虚拟机，我们就可以不断改进它：我们可以添加更多的指令、更多的寄

存器、不同的字长大小，或任何我们可能想尝试的东西。由此产生的 TOY 虚拟机比原来的 TOY 机器“更强大”。这个想法被称为步步为营法（bootstrapping）——一旦我们建立一台机器，我们可以用它来创建“更强大”的机器。这个基本理念在数十年来计算机的设计中发挥了至关重要的作用。有句引人瞩目的名言（当然，也有可能是杜撰的）就很好地传达了这样的想法：Cray Research 的创始人、几代超级计算机之父 Seymour Cray 听说苹果公司已经买了一台 Cray 来模拟计算机的设计。Cray 感到很开心，他说：“有意思！我正在用苹果来模拟 Cray 3。”

971

还有一个值得深思的问题：TOY 是否存在？虽然 TOY 机物理意义上并不存在，但是我们可以执行和调试 TOY 程序，而不依赖于是否存在物理上的 TOY 机器。事实上，从这个意义上来说 TOY 和 Java 没有区别。我们实现和调试 Java 程序，但没有物理 Java 机存在。实际上，程序 6.4.1 证明了 TOY 和 Java 一样真实。可能有数十亿台实现 Java 虚拟机的真机，其中每一台都实现了 TOY。换句话说，就 Java 的存在范围而言，TOY 也存在。Java 运行在数十亿台设备上，TOY 也一样。

上一节中我们讨论过的程序都可以很容易地存储在文件中（事实上，任何 TOY 程序都这样），而任何数据都可以经由标准输入读取。因此，在程序 6.4.1 中，构造函数将程序加载到 TOY 虚拟机中，从标准输入上获取信息，并模拟其产生的操作，并将其输出（如果有的话）呈现在标准输出上。

更重要的是，任何人（包括你）都可以实现自己设计的机器并为其编写程序。这是一个非常强大的创意，它将我们的计算基础设施带到了现在的状态，并将带领我们走向未来。

**TOY 系列虚拟计算机** 我们的虚拟机器只有 256 个字的内存，每个 16 位。尽管我们已经证明，原则上我们可以在 TOY 中开发任何可以用 Java 开发的程序，但是你通常会产生直接反应就是，TOY 没有足够的内存来做任何一个实际应用中重要的计算任务。

但是这个想法是完全错误的。像 TOY 这样的计算机在被引入之后的几年中被用于各种重要的应用。例如阿波罗导航计算机（Apollo Guidance Computer），六次成功完成登月导航任务，它只有 1024 个 16 位字的内存，只相当于 4 个 TOY 机！

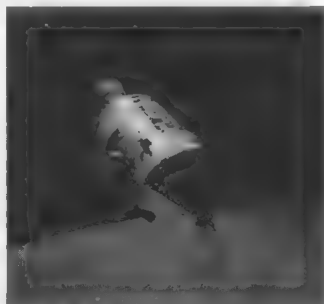
972

外部存储设备正在不断改进，从穿孔的纸带到穿孔卡，再到磁带和磁盘存储器，程序员们开始意识到，他们通过将程序分阶段地存储进内存，就可以使用一个相对较小的（但昂贵的）内部存储器来完成他们的工作，然后从外部存储器中读取下一个阶段的代码。到了 20 世纪 70 年代，这种观点引出了虚拟内存（virtual memory）的概念，操作系统为程序制造了一种“错觉”，让它“拥有”了比机器物理内存大得多的可用内存。这个想法在现代计算中仍然起着核心作用。

不过，随着技术的进步，我们的内存不断扩大，机器不断变快。现代计算机拥有数十亿位的内存。他们怎样与我们的 TOY 机器联系起来？

可以肯定的是，各方面的技术进步是不可思议的，但最重要的一点是，现代计算机上的机器语言程序的基本性质与 TOY 编程之间的差异比你想象得小得多。

TOY-64。为了将 TOY 与现代计算机联系起来，我们设想一个 64 位的 TOY 机器，我们称之为 TOY-64。对于这样的机器，虽须配备更多的二进制位来指定寄存器和存储器的位置，



照片来自美国宇航局

在月球轨道上的阿波罗 17 号

但其指令与我们之前讲述的那些机器完全一致。具体来说，我们可以把 40 位用于存储器地址，20 位用于寄存器地址。这意味着 TOY-64 可以容纳超过 680 亿个 64 位字，并且使用超过 250 000 个寄存器，这些寄存器当然比 TOY 或 PDP-8 更接近当代的计算机。



一个假想的 64 位计算机

这个机器的编程和 TOY 的编程是一样的，除了需要处理更大的内存字，更多的寄存器和更多的内存以外。即使我们没有展开过多的细节，你也可以看到，所有的东西都会用 16 位十六进制数字来表示，我们分析过的所有表示方法都可以自然地扩展。例如，数字 40544F592D363421 可以表示数字  $4635417160900293665_{10}$ 、字符串 “@TOY-64 !”，或者一个指令，它将寄存器 592D3 和寄存器 63421 的值按位异或，并将结果存储在寄存器 0544F 中。很容易看到我们如何将一个为 TOY 开发的程序转换成一个可以在 TOY-64 上运行的程序。

973

因为这种转换非常容易，在构建新的计算机时，能够大量快速地使用旧版软件就成为一个重要的优势。事实上，我们今天使用的大量软件都是几十年前开发的，但我们依然可以使用它们的原因是，新机器保持与旧版本的兼容性。

TOY-64 很可能没有开关和指示灯，只有一个无线接口和一个开 / 关按钮。技术细节并不重要。重要的是从程序的角度来看，机器可以访问长度不受限的输入 / 输出流。

TOY-64 和现代计算机最显著的区别是指令集。典型的机器将更多的位用于操作码，从而支持更丰富的指令来执行硬件上的任务，从浮点操作到存储器操作，再到外部存储器支持。然而，硬币具有两面性：这样任意扩展硬件指令使得软硬件开发都相对容易，但与之相对应的开发可靠的高性能软件很难（译者注：种类复杂的指令集不容易优化，也不容易使用流水线和指令并行等技术），因此发展出了精简指令集计算（RISC），并持续应用至今。典型的现代计算机可能比 TOY-64 指令多 2~4 倍，但不会比这更多。

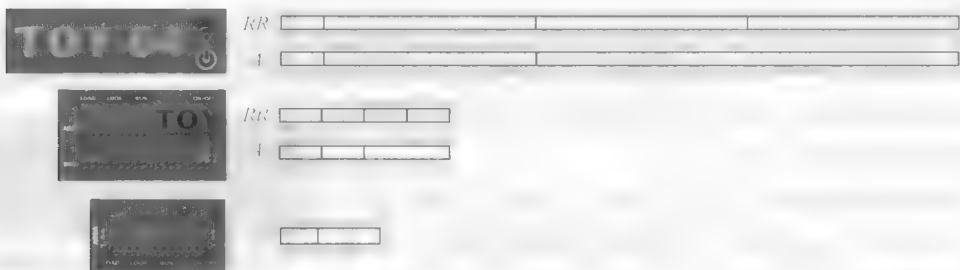
TOY-64 和现代计算机之间另一个显著的区别是，现代计算机没有太多的寄存器，或像 RR 那样的指令。我们不再详述这里的差异，需要说明的是所有计算机都有一个复杂的存储体系结构，从价格昂贵、容量小、速度快的存储到价格便宜、速度慢但容量大的存储设备。我们的寄存器表达的是一种设计思路，任何计算机体系结构必须考虑到内存技术的差异，因此我们需要一个处理器内部的中转存储。

这种情况与我们的模块化编程的“客户程序-API-实现模型”有点不同。软件和硬件之间的界限应该在哪里？很多计算机已经可以提供与 TOY 非常类似的编程接口，你随时可以开始在其上编写程序，所需的努力可能与学习一种新的编程语言类似。

|            | TOY | TOY-64         | TOY-8 |
|------------|-----|----------------|-------|
| 每个字长的位数    | 16  | 64             | 8     |
| 寄存器数量      | 16  | 262 144        | 1     |
| 内存中的字数     | 256 | 68 719 476 736 | 32    |
| 每个操作码的位数   | 4   | 4              | 3     |
| 每个寄存器地址的位数 | 4   | 20             | 0     |
| 每个内存地址的位数  | 8   | 40             | 5     |

TOY 系列产品的参数

974



TOY 系列虚拟计算机

TOY-8。为了能够将 TOY 与计算机的电路实现相关联，我们还在第 7 章中假想了一个 8 位的 TOY 机器，它配有 32 字节的存储器和一个寄存器，我们称之为 TOY-8。为这样的机器编写程序似乎是一个挑战，但是请注意，我们在本节中分析过的所有程序所需的内存都没有超过 32 字节。当你意识到一个程序可以从纸带上读取更多的代码时，你可以看到，实际上用这样一个微小的机器完成一些艰巨的任务是有可能的。

一个明显的事实是：即使在 TOY-8 中，存储器的可能状态的数量也已经达到 2256 个，这甚至还没有考虑外部存储器。因此，我们永远无法知道 TOY-8 可以做什么（当然，绝大多数 TOY-8 程序将永远不会真实存在）。

TOY-8 的意义在于展现一款与 TOY 具有几乎完全相同特性的完整机器，只是各类资源都少了一些。我们设计 TOY-8 的目的是为了能够展示一个包含 TOY（其实其他计算机也几乎一样）所有基本元素的计算机的完整电路。通过了解 TOY 编程的特点，可以理解 TOY-64 等计算机上的程序以及你自己的计算机程序的运行情况。通过了解 TOY-8 如何构建，可以想象出如何构建 TOY、TOY-64 以及你自己的计算机。

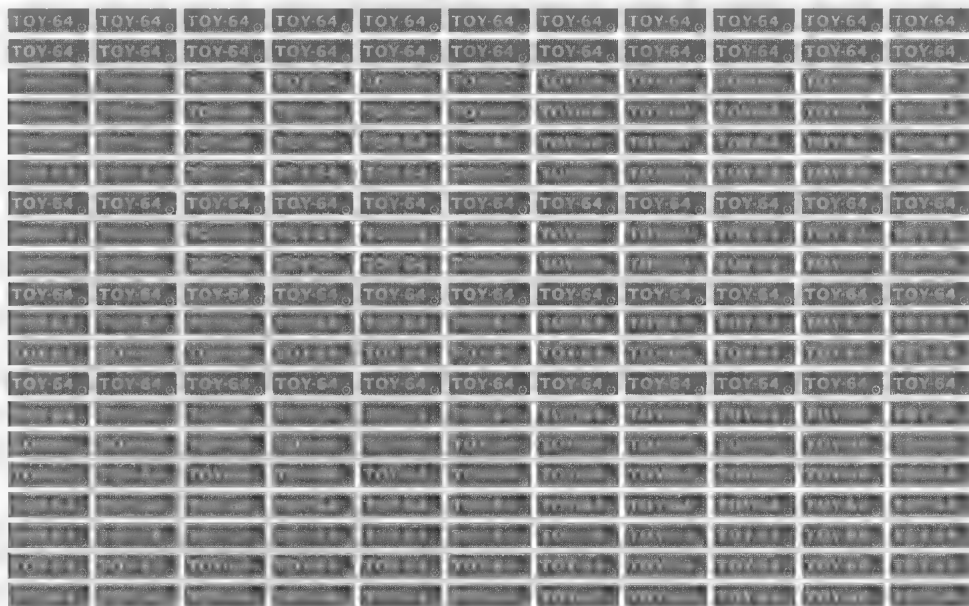
上面表格展示了这一系列虚拟计算机的各种参数。当然，TOY 和 TOY-64 之间还有很大差距，因为我们没有详尽分析 32 位计算机的所有特性，而这些特性是在过去的几十年中随着 32 位计算机的广泛使用不断丰富起来的。我们会在练习中涉及更多相关信息。

975

虚拟内存。看起来 TOY 最大的限制之一就是它的内存有限，所以你可能会想知道如何模拟比我们实际机器上的内存更多的内存。这个问题很早就被解决了，因为内存的成本一直很高，我们总是没有足够的内存可以使用。使用纸带有点难以想象，但是在很短的时间内，磁带和磁盘就出现了，并可以提供大量的外部存储器。有了这样的设备，虚拟内存（virtual memory）的想法很快就出现了。大多数程序在给定的时间只能处理相对较小的内存区域，因此程序可以访问大型虚拟内存，而实际上虚拟内存大部分是驻留在外部存储上的。操作系统的工作是确保程序需要访问的内存部分在适当的时间被放到实际的内存，以保证程序可以访问到它们。在这种情况下，可用的实际内存越多，程序越能更好地工作，毕竟在实际内存和外部存储间的交互越少，性能会越高。

向后兼容。我们今天所使用的大量软件，是很久之前在一台功能更少的机器上写就的，并经过了很多代的更新。这是个令人瞩目的事实。事实上，关于软件，除去它提供的功能之外，人们对于它一无所知。当你购买一台新计算机时，你使用的软件只有一小部分是那台计算机编写的，剩余的软件均来自以前的开发成果。这极大地加快了进度：如果虚拟机能够工作，那么所有的软件都能正常工作！但过了一段时间，可能会出现一些意想不到的问题。一个著名的例子就是 Y2K 问题，即公元 2000 年前开发的各種旧软件系统必须在跨千年的时候重写，因为它们仅用两位数字代表年份（的后两位），导致出现各种未预想到的后果，如年龄可能变成负值等。

服务器农场。为什么只使用一台计算机来完成计算任务？我们在程序 6.4.1 中实现的 TOY 只是一种数据类型，所以我们可以很容易地编写一个客户程序，可以创建一千或一百万个 TOY 计算机并运行它们。现代云计算使得它们可以在由大量实际处理器组成的服务器农场中同时运行。实际上，越来越多重要的计算应用需要在一个服务器农场的虚拟机环境中运行。当你用移动设备识别语音或增强照片时，很可能是服务器农场中的虚拟机帮助你完成了这项工作。 [976]



一个 TOY-64 服务器农场

我们设计了一台虚拟计算机，当然这个过程对于已经设计完成的计算机设备没什么贡献。但是这台计算机足够简单，可以使更好地理解计算的本质，从而对我们在第 5 章中学习的基本理论问题形成更深刻的理解。计算机究竟是什么？计算机程序究竟是什么？计算机的哪些方面体现了它的本质？有没有新的计算方法可以显著提高我们的计算能力？仿真似乎能够将如此广泛的各种不同的计算设备整合到统一的整体中，是否对其存在一些基本的解释？要找到这些问题的答案，你可以从安装器、解释器、编译器和其他类似的处理程序的程序中找到一些启示。

最后，我们需要通过研究如何设计和实现它的电路来揭开 TOY 本身的神秘面纱。这是下一章的主题。 [977]

## 问答环节

问：为什么不在程序 6.4.1 中使用 short 类型的数组来定义 R[] 和 M[]？它们只有 16 位，与 short 类型是一致的。

答：在使用算术运算符时，Java 会将 short 值补充完整为 int 值，因此这么做没有必要。但是，我们有时候需要用 0xFF 来对 int 型进行取掩码得到一个 16 位的值，以便模拟一个 short 值的行为。

问：如果将 PC 设置为 FF，会发生什么情况？在 FF 中是什么值？

答：当然，我们可以为这个操作定义我们想要的任何行为，但要回答这些问题，必须研究程序 6.4.1 的代码，因为这是 TOY 的定义。显然，M[FF] 最初是零，但是它保存了用 stdin() 读取或用 stdout() 写入的最后一个值。当然，某些黑客可能会利用这一行为！因此最好在 stdin() 和 stdout() 的实现中将其设置为零，这样当 PC 跳转到 0xff 时机器会停机。



问：TOY 是否通用（是否等价于图灵机）？

答：并不完全是的。如我们所定义的那样，用读写双向纸带取代纸带打孔器和读取器就可以了。事实上，这也是许多旧计算机第一次升级的项目之一。

## 练习

6.4.1 本章末尾描述的 TOY-64 服务器农场中有多少字节的内存？

6.4.2 实现 TOY.java 的 main() 方法（程序 6.4.1）。

答案：

```
public static void main(String[] args)
{
 String filename = args[0];
 int pc = Integer.parseInt(args[1], 16);
 TOY toy = new TOY(filename, pc);
 toy.run();
}
```

6.4.3 为 TOY.java 添加一个命令行参数，表示一个内存地址和一个寄存器号，然后添加代码，使得每次执行这个内存地址的指令之前（即 PC 取这一参数值），将指定寄存器的内容打印出来。如果参数为 0，则以正文中形式（在 5.3.1 节的开头处）打印程序结束时内存的内容。

6.4.4 为模拟穿孔纸带的 TOY.java 开发 stdin() 和 stdout()：每个 16 位值表示为两行，每行 8 个字符，其中“0”用空格表示，“1”用“\*”表示。

6.4.5 修改 TOY.java 用乘法指令替换减法指令。请注意两个 16 位整数相乘的结果是一个 32 位整数。

6.4.6 修改 TOY.java 以支持一个 216 字的 TOY 内存，并将每个内存相关指令的含义改为间接指令，其访问的目标地址放在前 256 个内存字中。例如，指令 8A23 会将 M[23] 中的内容当作地址，从内存中加载一个字长到 R[A] 中。为这台机器实现一个 sum.toy，用于实现 10 000 个 16 位二进制补码值的求和计算。

## 创新练习

6.4.7 单寄存器机器。设计一个 16 位计算机，使其具有一个寄存器、16 条指令和 4096 个字的存储器。对于每个双操作数指令，一个操作数放在寄存器中，另一个放在存储器中，并将结果留在寄存器中。为你的机器写一个模拟器。

6.4.8 虚拟内存。假设机器有一个新的外部存储设备，有 232 个可寻址的 32 位字，并且它与 TOY 相连接实现了一个虚拟内存，如下所示：对 FF 的两个连续写入提供一个 32 位地址，然后，如果接下来的两个指令是对 FF 的存储指令（或间接存储指令），则它们等同于向该地址写入一个 32 位值；如果接下来的两个指令是对 FF 的加载指令（或间接加载指令），则它们等同于从该地址读取一个 32 位值。写一个 sum.toy 的版本，可以为一百万个 32 位二进制补码值求和。

6.4.9 并行 TOY。修改 TOY.java 从命令行取整数  $n$ ，然后模拟一台维护  $n$  个 PC 的机器，编号分别为从 0 到  $n-1$ 。机器在每个周期内同时对所有 PC 模拟其提取-递增-执行的过程。如果一个周期中两个 PC 要求对同一个寄存器进行不同的更改，则索引较低的 PC 优先。

6.4.10 TOY 中的 TOY。在 TOY 中开发一个实现 TOY 虚拟机的程序 TOY.toy。首先假定机器有 32 个字的内存、8 个寄存器且没有标准输入/输出。可以使用其余的内存（和标准输入/输出）来开发你的程序。然后为其添加标准输入/输出、更多的内存和更多的寄存器，以便可以运行 6.3 节中的所有程序。

6.4.11 字符串 TOY。为一个假想的 16 位字符串处理机器设计和构建一个模拟器，它具有与 TOY 相



同的寄存器和内存，但是具有字符串处理操作。假定字符串被存储为一个内存字的序列，每个字用于存储两个 ASCII 字符，以 00 结尾。该计算机应支持字符串搜索、子串提取和标准输入/输出操作。编写一个程序来使用插入排序以排序一个字符串数组（数组中的每个元素是一个字符串的引用）。

980

- 6.4.12 性能。TOY 比 Java 更快吗？分别用 TOY 程序和 Java 程序实现一个使用 BST 为随机的 16 位整数去重的功能，运行倍速测试来观察两个程序的运行时间加速比率的差异。
- 6.4.13 汇编器。编写一个 Java 程序，用于处理由汇编语言编写的 TOY 程序（汇编语言是比机器指令稍微高级一些的编程语言），并输出一个 TOY 程序，采用的格式应适合于正文中给出的引导程序所需的格式。汇编语言可以为地址、操作码和寄存器定义符号名称。例如，以下是程序 6.3.4 的汇编语言版本（在标准输出上打出斐波那契数列）：

```

LA one, 1
LA a, 0
LA b, 1
L i, N
loop BZ i, done
ST a, stdout
A c, a, b
A a, b, 0
A b, c, 0
S i, i, one
BZ 0, loop
done H
N 000C

```

在实现汇编器时，你应该在汇编语言代码和 TOY 指令之间保持一对一的对应关系。其他具体细节由你来实现。汇编语言相对于机器语言的一大优点是程序可以在任何地方加载（程序中的地址由汇编器来计算），因此程序应该将起始地址作为命令行参数。

- 6.4.14 表达式编译器。修改 Dijkstra 算法（程序 4.3.5）输出一个可以计算给定表达式的 TOY 程序。
- 6.4.15 表达式解释器。开发 Dijkstra 算法（程序 4.3.5）的 TOY 实现，略过平方根函数。假设输入表达式在标准输入上满足：所有操作数都是非负的 15 位无符号数，使用负数来表示运算符和分隔符，如 8001 代表“+”、8002 代表“-”、8003 代表“\*”、8004 代表“/”、8005 代表“（”，8006 代表“）”。栈的实现参考练习 6.3.25，乘法运算实现参考练习 6.3.35。
- 6.4.16 32 位 TOY。设计一个 32 位的 TOY 计算机。论证你所做的每一个设计选择的合理性，为你的 TOY-32 设计实施一个虚拟机。

981

- 6.4.17 弹跳球。开发一个 TOY 程序，为一台绘图机产生指令。具体来说，假设一个采用 16 位命令的绘图机的指令格式是：第一个十六进制数字是操作码（4 位二进制数）；后面的数字是参数信息。对于这个练习，我们只关注两个操作码：0 操作码将一个字中剩余的 12 位值推进栈中，1 操作码从栈中弹出三个 12 位值（分别是  $r$ 、 $y$ 、 $x$ ），然后绘制一个以  $(x, y)$  为中心、 $r$  为半径的圆。例如，右侧的代码会在标准输出上产生一系列指令，可以用来指示设备绘制一个移动的球（从左到右移动，遇到右边界后弹回到左边）。参照程序 1.5.6，扩展这个程序来产生绘制弹跳球的指令。

```

10: 7AEE x
11: 7BFF y
12: 710F dx
13: 7C32 r
14: 9CFF push r
15: 9BFF push y
16: 1AA1 x += dx
17: 831E R[3] = mask
18: 3AA3 x &= mask
1A: 9AFF push x
1B: 871D R[7] = instruction
1C: 97FF push instruction
1D: C014 loop
1E: 1010
1F: 0FFF

```

为绘图设备创建指令

- 6.4.18 虚拟绘图机。编写一个 Java 程序 DrawingTOY.java，它使用 StdDraw 来模拟上一个练习中描述的绘图设备，以生成指定的动画。扩展机器以支持正方形、线条和多边形，然后编写 TOY 代码以生成有趣的图形设计。

982  
983

## 构建计算设备

如果要设计一个计算机处理器，你可能会想，这需要一支受过最专业培训的高级人才队伍。虽然这个想法有些道理，但是自第一台计算机以来，在所有处理器的整体架构设计的过程中有着显著的简易性和相似性，本章将会详细地讲述一台特殊的通用计算机的设计过程。通过讲述计算机的设计，解释“如何构建计算机”“如何操作计算机”诸如此类的问题。

通常，我们可以将计算机视为连接到输入和输出设备的黑匣子。那么里面是什么？如果打开计算机，你可能会看到一些通过插入电路板连接在一起的模块，它们中的大部分的作用就是控制输入和输出设备，其中一个模块是中央处理器（CPU），它是计算机的心脏、思想和灵魂，因为如果没有 CPU 所发出的信号，整个计算机将无法工作。如果我们可以通过显微镜看到 CPU 的内部结构，你会看到它基本上就是一个由若干模块连接在一起而组成的内部网络。这和早期的计算机有很大的不同，由于电池和电线的体积较大，早期计算机甚至会填满一间屋子。

本章使你了解计算机如何工作，甚至可以自己设计一台计算机。如同计算机科学中的许多项目一样，你需要注意细节，但计算机设计的概念十分简单，我们会利用一种抽象的思想来解决这个问题。

984  
985

### 7.1 布尔逻辑

我们对数学函数的概念已经很熟悉了——我们在程序的执行中就讨论过它们。布尔函数也是将参数映射到一个值的数学函数，只是其中范围（函数参数）和值域（函数值）都仅仅是两个值中的一个。无论我们将这两个值称为 true 和 false、是与否，还是 0 和 1，概念都是一样的。布尔函数的研究被称为布尔逻辑（Boolean logic）。

布尔逻辑得名于 19 世纪的英国数学家乔治·布尔（George Boole）。从那以后，它成为逻辑推理的基础。如果你希望对布尔逻辑展开深入的研究，你需要另外找一本书（研究这样一本书或者就这个问题选修一门课程将都是非常有价值的）。在本节中，我们从基础理论开始，重点关注与计算相关的概念，特别是数字电路的实现。

我们在本书中遇到过几次布尔函数。这里只举几个例子：

- 第 1 章介绍了 Java 的布尔值数据类型，并立即学会在程序中的 if 和 while 语句中使用它来实现决策。
- 在第 5 章中，我们考虑了布尔可满足性问题在计算理论中的关键作用。
- 在第 6 章中，我们看到了布尔函数在对信息进行二进制表示时的实用性。



乔治·布尔（1815—1864）

由于它的重要性，我们在本节中加入了“前文索引”，所以如果你只看了本章，而没有阅读本书的其余部分，也不会有问题，因为涉及的前文中的基本信息并不难理解，你仅阅读本章也不影响你掌握其中的知识。我们之所以选择这种非线性结

构，是因为布尔函数和执行计算任务的电路之间存在的密切联系。这是一个基础的概念，我们基于它发展出了今天所用的计算机基础设施。本章希望你重点关注于布尔函数和电路的关系，同时更好地了解布尔对数学的伟大贡献。他完全没有料到他的工作将在两个世纪后作为计算的基础。理解布尔函数和电路的关系等同于理解“电路如何计算”的问题。

986

**布尔函数** 你所熟悉的布尔函数包括非、或、与的基本功能，所以我们从它们开始。要定义任何布尔函数，我们只需要为其每个可能的输入值确定其输出值。在本节中，我们使用 0 和 1 表示布尔值。我们使用布尔变量（boolean variable）在符号表达式中表示布尔值。例如，not 函数是一个布尔变量的函数，定义如下：

$$\text{NOT}(x) = \begin{cases} 0 & \text{如果 } x \text{ 是 } 1 \\ 1 & \text{如果 } x \text{ 是 } 0 \end{cases}$$

类似地，包含两个变量的函数与（and）、或（or）、异或（exclusive or）定义如下：

$$\text{AND}(x, y) = \begin{cases} 0 & \text{如果 } x \text{ 或 } y \text{ (或两者) 为 } 0 \\ 1 & \text{如果 } x \text{ 和 } y \text{ 均为 } 1 \end{cases}$$

$$\text{OR}(x, y) = \begin{cases} 0 & \text{如果 } x \text{ 和 } y \text{ 均为 } 0 \\ 1 & \text{如果 } x \text{ 和 } y \text{ (或两者) 为 } 1 \end{cases}$$

$$\text{XOR}(x, y) = \begin{cases} 0 & \text{如果 } x \text{ 或 } y \text{ 相同} \\ 1 & \text{如果 } x \text{ 和 } y \text{ 不同} \end{cases}$$

直观地理解这些定义的一个方法是将  $x$  和  $y$  解释为逻辑命题，如“天空是蓝色的”或“太阳照耀着大地”，然后将 1 解释为真，将 0 解释为假。例如，如果  $x$  和  $y$  都为 true，则  $\text{AND}(x, y)$  为 true，否则为 false，这恰好支持了我们的直觉，即“天空是蓝色的，并且太阳照耀着大地”这样的语句只有在命题的两个部分都是真实的，才能判定为真。在数学定理中，如果  $x$  是“整数  $v$  大于或等于 0”的命题， $y$  是“整数  $v$  小于或等于 0”的命题，则  $\text{AND}(x, y)$  为真也就是说  $v$  等于 0（符合有关整数的相关公理）。这种应用是布尔研究这些功能的动机。

表示形式。在近两个世纪中，布尔逻辑已经在多个领域发挥了重要的作用，因此对于基本操作也出现了许多不同的表示形式。我们已经在本书遇到了一些。在本章中，我们使用  $x'$  表示  $\text{NOT}(x)$ ，用  $xy$  表示  $\text{AND}(x, y)$ ，用  $x + y$  表示  $\text{OR}(x, y)$ 。用乘法来表示“与”操作与我们之前的认知相符，但是用加法表示“或”操作会有一个特殊的表达式： $1 + 1 = 1$ 。我们在本章中会经常使用它，但是为了防止混淆，我们在以下表格中总结了所有的表达式。

987

|     | 逻辑           | Java布尔值          | Java位运算          | 电路设计         |
|-----|--------------|------------------|------------------|--------------|
| NOT | $\neg x$     | $!x$             | $\sim x$         | $x'$         |
| AND | $x \wedge y$ | $x \ \&\& \ y$   | $x \ \& \ y$     | $xy$         |
| OR  | $x \vee y$   | $x \ /\ / \ y$   | $x \   \ y$      | $x + y$      |
| XOR | $x \oplus y$ | $x \ \wedge \ y$ | $x \ \wedge \ y$ | $x \oplus y$ |

基本的布尔函数表达式

**真值表。**我们已经注意到，定义布尔函数的一种方法是为输入参数的每种可能情况定义其输出值。这样的数据的一种有效组织形式是真值表。在一个真值表中，每个变量对应一列，变量值组合的一种可能情况对应一行，最后一列用于该变量组合对应的函数值。例如，

下面列出了一些基本函数的真值表定义：

| NOT |    | AND |   |    | OR |   |     | XOR |   |     |
|-----|----|-----|---|----|----|---|-----|-----|---|-----|
| x   | x' | x   | y | xy | x  | y | x+y | x   | y | x+y |
| 0   | 1  | 0   | 0 | 0  | 0  | 0 | 0   | 0   | 0 | 0   |
| 1   | 0  | 0   | 1 | 0  | 0  | 1 | 1   | 0   | 1 | 1   |
|     |    | 1   | 0 | 0  | 1  | 0 | 1   | 1   | 0 | 1   |
|     |    | 1   | 1 | 1  | 1  | 1 | 1   | 1   | 1 | 0   |

基本布尔函数的真值表

有  $n$  个变量的函数，它的真值表有  $2^n$  行，所以我们不能用真值表表示过多的值。如下图所示，我们使用真值表不仅仅是定义函数，还可以验证它们的各种操作和应用的有效性，因为它们为我们提供了一种系统的方式来检查所有的可能性。

988

两个变量正好有 16 种布尔函数，所以我们可以枚举它们，如下表所示：

| x | y | 0 | AND | xy' | x | y | XOR | OR | NOR | EQ | y' | x' | NAND | 1 |
|---|---|---|-----|-----|---|---|-----|----|-----|----|----|----|------|---|
| 0 | 0 | 0 | 0   | 0   | 0 | 0 | 0   | 0  | 1   | 1  | 1  | 1  | 1    | 1 |
| 0 | 1 | 0 | 0   | 0   | 0 | 1 | 1   | 1  | 0   | 0  | 0  | 1  | 1    | 1 |
| 1 | 0 | 0 | 0   | 1   | 1 | 0 | 0   | 1  | 0   | 0  | 1  | 0  | 0    | 1 |
| 1 | 1 | 0 | 1   | 0   | 1 | 0 | 1   | 0  | 1   | 0  | 1  | 0  | 1    | 0 |

两个变量的所有布尔函数

我们经常在数学逻辑、数字电路设计的过程中遇到这些函数，甚至没有标记的那几列也属于常见的操作（见练习 7.2.2）。特别要说明的是，你需要注意 NOR（NOT OR）、NAND（NOT AND）和  $xy'$ （AND NOT）的真值表。

布尔代数。布尔运算符（boolean operator）是表示布尔函数的符号；布尔代数是指由布尔变量和布尔运算符组成的表达式的符号操作。你将看到，布尔代数最大的限制就是变量取值只能为 0 和 1，这使得布尔代数与你在中学学到的实数代数不同（也简单得多），但也有很多相似之处。

代数作为广义数学的研究对象，是一个普遍的概念，它也是一个更高级别的主题，已经超过了我们研究的范围。对于布尔代数，我们首先给出对公理（公认的事实）的严谨定义，然后在它的基础上从逻辑上推断布尔函数的一些定理和公式。为了表达方便，我们将公理、公式和定理都称为“定律”，你可以利用其中的任何一个。布尔代数的基本定律的定义非常简单，其中有很多你都很熟悉。在代数中常用到的交换律、分配律和结合律在布尔代数中仍然适用，如下表所示。此外，很多其他仅适用于布尔代数的定律也很容易证明。例如，表中的最后一项给出了 NAND 和 NOR 函数的两个特殊等式，我们称之为德·摩根定律。

989

|     |                 |
|-----|-----------------|
| 公理  | $1x = x$        |
| 恒等式 | $x + 0 = x$     |
|     | $xx' = 0$       |
| 推论  | $x + x' = 1$    |
|     | $xy = yx$       |
| 交换律 | $x + y = y + x$ |

布尔代数的基本定律

分配律

$$x(y + z) = xy + xz$$

$$x + yz = (x + y)(x + z)$$

结合律

$$(xy)z = x(yz)$$

$$(x + y) + z = x + (y + z)$$

公式和定理

取反

$$0' = 1$$

$$1' = 0$$

双重否定

$$(x')' = x$$

0-1律

$$0x = 0$$

$$1 + x = 1$$

吸收律

$$x(x + y) = x$$

$$x + xy = x$$

德·摩根定律

$$(xy)' = x' + y'$$

$$(x + y)' = x'y'$$

布尔代数的基本定律 (续)

990

所有这些定律都很容易建立真值表。我们在 1.2 节的一个例子中已经使用过这种方法。由于它们的重要性，我们使用本章的符号来重新表述德·摩根定律的真值表：

| NAND     |          |           |              |  | NOR      |          |            |               |  |
|----------|----------|-----------|--------------|--|----------|----------|------------|---------------|--|
| <i>x</i> | <i>y</i> | <i>xy</i> | <i>(xy)'</i> |  | <i>x</i> | <i>y</i> | <i>x+y</i> | <i>(x+y)'</i> |  |
| 0        | 0        | 0         | 1            |  | 0        | 0        | 0          | 1             |  |
| 0        | 1        | 0         | 1            |  | 0        | 1        | 1          | 0             |  |
| 1        | 0        | 0         | 1            |  | 1        | 0        | 1          | 0             |  |
| 1        | 1        | 1         | 0            |  | 1        | 1        | 1          | 0             |  |

证明德·摩根定律的真值表

Java 中的布尔代数。你可能已经意识到了，Java 程序中的布尔代数有两种不同的形式。两者之间的区别是新手混乱的原因（也常是教师考试题材的来源），所以值得研究。

- Java 的布尔数据类型：在 1.2 节中，我们引入了布尔类型（取值范围为 true 和 false）和布尔运算，分别使用运算符 &&、||、! 进行 AND、OR 和 NOT 运算。自此以来，我们一直使用布尔表达式来控制程序中的执行流程，其中布尔表达式可以包括类型为 boolean 的变量、这些布尔操作符，以及可以返回布尔值的其他各类运算符。Java 支持任意的布尔表达式，如我们的第一个示例 LeapYear（程序 1.2.4）所示，但是其余程序几乎在所有情况下都使用了尽可能简单的表达式。
- 整数值的按位操作：在 6.1 节中，我们讨论了 Java 的按位操作，分别使用运算符 &&、||、! 和 ^ 对整数值的二进制表示形式的每一位进行 AND、OR、NOT 和 XOR 运算。Java 支持任意形式的按位操作表达式，这一点对我们处理数据的单个位很有帮助。在虚拟机 TOY（程序 6.4.1）中可以看到各式各样的应用。

每一个程序员在使用布尔代数时可能会用到以上两种方式的任何一种，这些操作在现代编程语言中是一个重要的组成部分。

**应用实例** 下面我们来分析一个加密的应用程序，这是把上述概念付诸实践的一个具体实例。

密码学中的根本问题是，发送者（sender）需要以窃听者（eavesdropper）不能够读取的

991

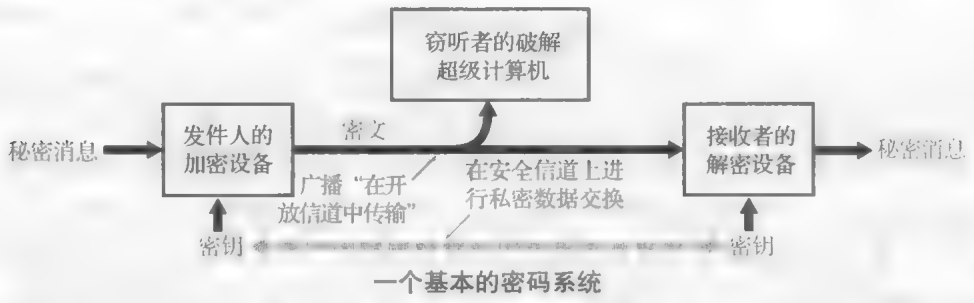
方式，向接收者（receiver）发送秘密消息。为此，一种简单的配置是发送者使用加密设备（encryption device）来创建密文（将要传递的信息编码后得到）并传输，而接收者则使用解密设备（decryption device）对其进行解码。密码系统（cryptosystem）即是解决这类问题的协议。

“一个基本的密码系统”图示了一个基本的密码系统。它基于使用密钥加密的方法来实现安全通信。这个方法让发送者和接收者通过一些安全机制提前交换密钥，以便发件人可以使用密钥加密消息，而接收者可以使用相同的密钥对其进行解密。例如，在 20 世纪的世界大战中，指挥官和船长将采用与总部使用相同的“密码本”，每天都会有当天的密钥，以应用于当天的安全通信。

一种特别简单的加密 / 解密方法即是布尔逻辑的直接运用。要发送消息，先将其转换为二进制字符串，然后通过使用按位异或操作（XOR）加密以形成密文，如下所示：

| 消息      |   | S | E | C | R | E | T |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 消息（二进制） | m | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

加密密钥只是一个与消息等长的字符串。当然，加密“设备”现在只是一个程序，并且编写一个 Java 程序，以用于在任意长的两个二进制位序列上执行这个计算来产生一个密文，这并不是一个复杂的任务（见练习 7.1.18）。如果密钥的位是随机选择的，则密文不能被窃听者理解（其位也是随机的）。密码系统的安全性能高低取决于密钥随机的程度——密码学的科学性和艺术性即在于贡献尽可能多的随机性的密钥，以达到窃听者即便在满屋的超级计算机的帮助下也无法解密的程度。



992

但是接收方如何解密该消息呢？你可能会惊讶地发现，使用相同的密钥，接收方解密消息与发送者加密消息的过程是一样的：

|         |              |                                                  |   |   |   |   |   |  |
|---------|--------------|--------------------------------------------------|---|---|---|---|---|--|
| 消息      | c            | 100110100111100011111010001110011101110111101011 |   |   |   |   |   |  |
| 密钥      | k            | 110010010011110110111001011010111001100010111111 |   |   |   |   |   |  |
| 消息（二进制） | $c \oplus k$ | 010100110100010101000011010100100100010101010100 |   |   |   |   |   |  |
| 密文      |              | S                                                | E | C | R | E | T |  |

这个过程乍一看似乎很神奇，但是我们可以证明： $(c \oplus k) = ((m \oplus k) \oplus k) = m$ ，从而能够理解它的原理。这些是对消息的每一位执行的操作，因此接收者也需要对每一位进行恢复。

我们当然可以通过真值表来证明这种方法的正确性（参见练习 7.1.4），但它是一个用来说明布尔代数法的很好的例子，我们无须诉诸真值表即可证明这种正确性。下表给出了证明上述表达式正确性所需的代数定义和定律：

|     |                                                 |
|-----|-------------------------------------------------|
| 定义  | $x \oplus y = xy' + x'y$                        |
| 恒等律 | $x \oplus 0 = x$                                |
| 归零律 | $x \oplus x = 0$                                |
| 结合律 | $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ |

恒等律、归零律和结合律很容易从前文的布尔代数的定义和基本定律中得到证明（见练习 7.1.5）。根据这些定律，我们的证明过程就变得很简单：

$$\begin{aligned} (m \oplus k) \oplus k &= m \oplus (k \oplus k) && \text{结合律} \\ &= m \oplus 0 && \text{归零律} \\ &= m && \text{恒等律} \end{aligned}$$

993

这种非常简单的机制已经被广泛应用了很长时间，并且在现代系统中仍然被广泛使用。当然，现在的人们很少依赖可信的信使进行密钥交换——现代密码学基于更复杂和更方便的密钥分发方法（最广泛使用的方法被称为 RSA，这是一种基于因式分解困难性的技术）。当然，产生“随机”密钥仍然是一个热门的研究课题（简单的例子见练习 7.1.15）。

这个应用程序只是布尔代数效用的一个小例子。然而，正如我们在本书中看到的许多数学应用一样，这证明了基础研究的重要性。近两个世纪前，布尔为了自己的缘故追求知识，他绝没有想到他的代数对我们理解互联网商业周边的密码基础设施而发挥的作用。

**三个或更多个变量的布尔函数** 随着变量数量的增加，函数的可能性也会急剧增加。3 个变量有  $2^8$  个不同的函数，4 个变量有  $2^{16}$  个函数，5 个变量有  $2^{32}$  个函数，6 个变量有  $2^{64}$  个函数，等等。我们可以从这些数字中得出一个结论，正如我们在 5.5 节中有关指数增长的讨论一样，当布尔函数的变量数目较大时，其中的绝大多数布尔函数可能我们都不会遇到。但是有几个这样的函数在计算和电路设计中起着至关重要的作用，所以我们现在来讨论一下。

我们按照两个参数的方式可以很自然地扩展出多个参数的 AND 和 OR 函数的定义：

$$\begin{aligned} \text{AND}(x_1, \dots, x_n) &= \begin{cases} 0 & \text{如果任何参数为 0} \\ 1 & \text{如果所有参数为 1} \end{cases} \\ \text{OR}(x_1, \dots, x_n) &= \begin{cases} 0 & \text{如果所有参数为 0} \\ 1 & \text{如果任何参数为 1} \end{cases} \end{aligned}$$

在 AND 函数的真值表中，除最底部以外的所有值均为 0。在 OR 函数的真值表中，除最顶层之外的所有值均为 1。

994

可能性不胜枚举，事实上，布尔函数可以涵盖我们能够想到的任何计算任务。例如，我们可以定义一个布尔函数  $\text{PRIME}(x_1, \dots, x_n)$ ，当且仅当二进制数  $x_1, x_2, x_3, \dots, x_n$  是质数时，这个函数的值为 1。或者，我们也可以定义一个布尔函数  $\text{TOY}_k(x_1, \dots, x_n)$ ，函数值为 1 的条件是：当且仅当一个操作员初始化内存里的所有位，在  $x_1, x_2, x_3, \dots, x_n$  上加上 PC，然后摁下 RUN 键，TOY 机停机，并且前面那块控制板上从右边数第  $k$  位的指示灯亮起。这样的函数的定义可以是清晰而完整的，尽管它们的真值表可能会大到难以想象。

下面我们讨论在数字电路设计中出现的两个例子：majority (MAJ) 函数和 odd-parity (ODD) 函数：

$$\begin{aligned} \text{MAJ}(x_1, \dots, x_n) &= \begin{cases} 1 & \text{如果参数中 1 的个数比 0 多（不包括相等的情况）} \\ 0 & \text{其他情况} \end{cases} \\ \text{ODD}(x_1, \dots, x_n) &= \begin{cases} 1 & \text{如果参数中 1 的个数为奇数} \\ 0 & \text{其他情况} \end{cases} \end{aligned}$$



这些函数与 AND 和 OR 一样是无序的 (symmetric)，即它们的结果不依赖于它们的参数的顺序。

布尔表达式。与两个变量的布尔函数一样，我们可以使用一个真值表来指定任何布尔函数变量的值。对于三个变量，这样的表有  $2^3 = 8$  行。例如，这里是给出三个变量的 AND、OR、MAJ 和 ODD 函数的真值表：

| x | y | z | AND(x,y,z) | OR(x,y,z) | MAJ(x,y,z) | ODD(x,y,z) |
|---|---|---|------------|-----------|------------|------------|
| 0 | 0 | 0 | 0          | 0         | 0          | 0          |
| 0 | 0 | 1 | 0          | 1         | 0          | 1          |
| 0 | 1 | 0 | 0          | 1         | 0          | 1          |
| 0 | 1 | 1 | 0          | 1         | 1          | 0          |
| 1 | 0 | 0 | 0          | 1         | 0          | 1          |
| 1 | 0 | 1 | 0          | 1         | 1          | 0          |
| 1 | 1 | 0 | 0          | 1         | 1          | 0          |
| 1 | 1 | 1 | 1          | 1         | 1          | 1          |

995 三变量的布尔函数

对于具有较多变量的函数，用这种表示不仅麻烦而且容易出错，我们可以从图表中看出来， $n$  个变量需要  $2^n$  行。所以我们习惯使用布尔表达式来定义布尔函数。我们不难证明以下两个等式：

$$\text{AND}(x_1, \dots, x_n) = x_1 x_2 \dots x_n$$

$$\text{OR}(x_1, \dots, x_n) = x_1 + x_2 + \dots + x_n$$

从布尔代数的角度来说，我们可以用这些表达式来定义这些函数。但是，对应于 MAJ、ODD 这样的函数，我们应该怎么做？肯定还会有其他布尔函数难以写出其布尔表达式。

积之和表示。布尔代数的基本结论之一是，每一个布尔函数都可以使用操作符 AND、OR 和 NOT（对应的表示符号是 +、联结和 '）表示出来，成为一个布尔表达式。这个结论听上去或许会令人惊讶，稍后你会更加惊讶地实现，实现它们其实很简单。例如下面的真值表：

| x | y | z | MAJ | x' | y' | z' | x'yz | xy'z | xyz' | xyz | x'yz + xy'z + xyz' + xyz |
|---|---|---|-----|----|----|----|------|------|------|-----|--------------------------|
| 0 | 0 | 0 | 0   | 1  | 1  | 1  | 0    | 0    | 0    | 0   | 0                        |
| 0 | 0 | 1 | 0   | 1  | 1  | 0  | 0    | 0    | 0    | 0   | 0                        |
| 0 | 1 | 0 | 0   | 1  | 0  | 1  | 0    | 0    | 0    | 0   | 0                        |
| 0 | 1 | 1 | 1   | 1  | 0  | 0  | 1    | 0    | 0    | 0   | 1                        |
| 1 | 0 | 0 | 0   | 0  | 1  | 1  | 0    | 0    | 0    | 0   | 0                        |
| 1 | 0 | 1 | 1   | 0  | 1  | 0  | 0    | 1    | 0    | 0   | 1                        |
| 1 | 1 | 0 | 1   | 0  | 0  | 1  | 0    | 0    | 1    | 0   | 1                        |
| 1 | 1 | 1 | 1   | 0  | 0  | 0  | 0    | 0    | 0    | 1   | 1                        |

MAJ(x, y, z) 的积之和表示的真值表证明

从表中加粗显示的两列可以看出，对于每一种变量取值的情况它们的值都是相等的，所以我们证明了以下等式：

$$\text{MAJ}(x, y, z) = x'yz + xy'z + xyz' + xyz$$

我们构造的布尔表达式称为此布尔函数的积之和表达式或析取范式。

以下过程描述如何从真值表中得到布尔函数的布尔表达式：对于真值表中函数值为 1 的每一行，先找到一个变量的布尔乘积表达式，使得该表达式仅在这一行取值为 1，其他行均为 0；在得到每一行对应的小项之后，通过取不同小项的布尔和，就能构造出布尔表达式，使其输出值与布尔函数一致。每一个小项的表达式都是输入变量（假若这个变量取值对应

的那一行是 1) 或者是变量取反 (若其对应的那一行是 0) 的乘积。例如, 表达式  $xyz'$  对应 “1 1 0” 行, 因为当且仅当  $x$ 、 $y$ 、 $z$  的值分别为 1、1、0 的时候, 表达式的值为 1。你很容易可以想到, 这些小项的表达式之和就是这个布尔函数的表达式。

此方法适用于任何布尔函数。表达式中布尔积项的数量取决于函数值真值表中有多少项为 1。又如奇偶校验函数的真值表:

| x | y | z | ODD | $x' y' z'$ | $x' y' z$ | $x' y z'$ | $x' y z$ | $x y z'$ | $x y z$ | $x' y z + x y' z + x y z' + x y z$ |
|---|---|---|-----|------------|-----------|-----------|----------|----------|---------|------------------------------------|
| 0 | 0 | 0 | 0   | 1          | 1         | 1         | 0        | 0        | 0       | 0                                  |
| 0 | 0 | 1 | 1   | 1          | 1         | 0         | 1        | 0        | 0       | 1                                  |
| 0 | 1 | 0 | 1   | 1          | 0         | 1         | 0        | 1        | 0       | 1                                  |
| 0 | 1 | 1 | 0   | 1          | 0         | 0         | 0        | 0        | 0       | 0                                  |
| 1 | 0 | 0 | 1   | 0          | 1         | 1         | 0        | 0        | 1       | 1                                  |
| 1 | 0 | 1 | 0   | 0          | 1         | 0         | 0        | 0        | 0       | 0                                  |
| 1 | 1 | 0 | 0   | 0          | 0         | 1         | 0        | 0        | 0       | 0                                  |
| 1 | 1 | 1 | 1   | 0          | 0         | 0         | 0        | 0        | 1       | 1                                  |

ODD(x, y, z) 的积之和表示的真值表证明

当我们在设计计算任务的电路时, 还会遇到这个问题, 以及许多类似的问题。在本节的最后, 我们给出关于布尔函数的基本定理。

**定理 (Boole, 1847)** 任何一个布尔函数都可以表示为它的变量和相反值的积之和形式。

证明: 请参阅前文叙述

布尔逻辑为数字电路的设计奠定了基础, 这将在本章的后续部分深有体会。特别是当我们通过构建数字电路来计算布尔函数时, 积之和形式具有重大的意义。具体来说, 使用这种表达形式将为任一布尔函数构建数字电路的问题简化为构建实现 AND、OR 和 NOR 的电路组成问题。7.2 节中将介绍一些基础知识, 我们将在 7.3 节再次深入探讨这一主题。

997

练习

- 7.1.1 本节中讲到两个变量的布尔函数的布尔表达式表示时, 使用的表格中有若干列没有给出标签, 请补充完整 (例如表达式  $xy'$  是 AND 和 NOT 函数表示的标签)。
- 7.1.2 在德·摩根定律中, 如果 “或” 改为 “异或”, 还成立吗? 试证明或举出反例。
- 7.1.3 请用真值表证明  $x + yz = (x + y)(x + z)$ 。
- 7.1.4 请用真值表证明  $((m \oplus k) \oplus k) = m$ 。
- 7.1.5 从  $x \oplus y = xy' + x'y$  的定义和布尔代数的基本定理出发, 证明恒等律、归零律和结合律。(参考前文。)
- 7.1.6 使用正文中给出的密钥, 使用正文使用的方法, 给出消息 ANSWER 被编码时产生的密文。通过使用相同的密钥和相同的方法解密密文来检查答案。
- 7.1.7 证明  $MAJ(x, y, z) = xy + xz + yz$ 。
- 7.1.8 使用真值表证明:

$$MAJ(x, y, z) = (x + y + z)(x' + y + z)(x + y' + z)(x + y + z')$$

- 7.1.9 以上一题的表达式为例, 证明布尔函数可以表示为若干布尔和表达式的布尔积。这种表现形式被称为和之积表达式。

7.1.10 给出一个与  $\text{MAJ}(w, x, y, z)$  等价的布尔表达式。

7.1.11 对于一个三参数布尔函数，仅当  $xy = 1$  时，函数值为 1，其余情况为 0。写出其积之和表示。

7.1.12 对于一个三参数的 2 选 1 复合函数，即如果  $z$  为 0 则函数值为  $x$ ，如果  $z$  为 1 则函数值为  $y$ ，给出其积之和表达式。

998 7.1.13 为  $(x+y)(x+z)(y+z)$  函数列出其真值表，并写出一个等价的简化版表达式。

7.1.14 证明德·摩根规律可以扩展到  $n$  个变量，即，证明对于任意正整数  $n$ ，以下两个等式均成立。

$$(x_1 x_2 \cdots x_n)' = x_1' + x_2' + \cdots + x_n'$$

$$(x_1 + x_2 + \cdots + x_n)' = x_1' x_2' \cdots x_n'$$

999

## 创新练习

7.1.15 LFSR。用 Java 编写一个线性反馈移位寄存器 (Linear Feedback Shift Register, LFSR)，用来产生随机的序列位。该程序模拟下图所示的 12 位寄存器的操作。



寄存器计算第 11 位和第 9 位的异或，将结果写到第 0 位并把这 12 位的值输出，然后将所有位向左移一位。在程序中，使用 0 和 1 字符来表示位（类似练习 7.1.18 的方案），并使用以下方法：用一个由“0”和“1”组成的字符串作为命令行参数为这 11 位赋初值。然后第二个命令行参数为整数  $n$ ，并将以下操作重复执行  $n-11$  次来创建一个长度为  $n$  的字符：将第  $(i-11)$  与  $(i-9)$  位进行异或得到第  $i$  个字符。你的程序运行效果应该如下所示：

```
% java LFSR 11001001001 48
110010010011110110111001011010111001100010111111
```

答案：

```
public class LFSR
{
 public static void main(String[] args)
 {
 String fill = args[0];
 int n = Integer.parseInt(args[1]);
 for (int i = 11; i < n; i++)
 if (fill.charAt(i-11) == fill.charAt(i-9))
 fill += "0";
 else fill += "1";
 StdOut.println(fill);
 }
}
```

1000

7.1.16 有效 LFSR。练习 7.1.15 中给出的程序在  $n$  变得很大的时候将不再适用。它存在两个问题，一方面它运行需要指数级时间，另一方面在  $2^{11}-1$  位之后，序列将开始重复。我们将 11 和 9 替换为 63 和 62 来解决第二个问题，并且通过只保留前面打印出来的 63 位来解决第一个问题。

7.1.17 真值表。编写任何给定的布尔函数的客户程序，打印出该函数的真值表。用一个布尔值的数组作为函数唯一的参数来传递所需的所有输入信息。提示：请参阅 SATsolver（程序 5.5.1）。

7.1.18 加密 / 解密机器。开发一个 Java 程序 Crypto，从标准输入上读取两个长度相等的字符串，且两个字符串都是由 0 和 1 字符组成的，将这些字符看作二进制位，并将这两个输入字符串“按位异或”。也就是说，你的程序运行效果应该如下：

```
% java Crypto
010100110100010101000011010100100100010101010100
110010010011110110111001011010111001100010111111
100110100111100011111010001110011101110111101011
```

7.1.19 通用的基本函数集。假设有一组基本函数集合，通过集合中函数的组合可以实现每个布尔函数，那么就可以将该集合用硬件实现，从而得出任意函数的电路实现。我们将这个集合称为通用的基本函数集。本节中的积之和表明了 {AND, OR, NOT} 就是这样的通用函数（在练习 7.1.9 中的和之积也是这样的）。已知 NOT 只有一个参数，所有其他函数有两个参数，以下函数集里只有一个是通用的，其他函数集不是，找出它并证明你的结论。

- a. NOT and AND
- b. NOR
- c. NAND
- d. AND and OR
- e. NOT and OR
- f. AND and XOR

1001

7.2 基本电路模型

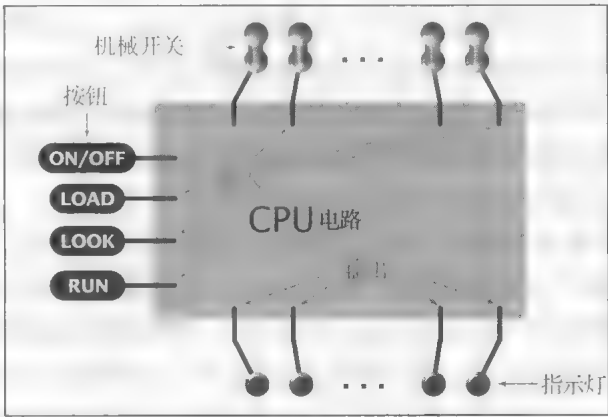
要了解计算机是怎么设计的，我们先来了解一下构造电路的三个基本元素：

- 导线
- 电源
- 控制开关

导线负责连接电源、传输数据，并连接电路元件；控制开关用于控制电路的闭合或是断开。一些导线被指定为输入；其他一些被指定为输出。我们的电路抽象定义如下：

电路是导线、电源和控制开关组成的互连网络，能够将输入线上的值转换为输出线上的值。

该模型足以描述任何计算设备。例如，我们可以使用它来描述第 6 章的 TOY 计算机的构造：TOY 机的 CPU 也是一个电路，其输入端连接着前面板上的机械开关以及按钮，输出端连接着前面板上的指示灯。我们的目标是设计一个电路，它能够根据开关设置和按钮在正确的时候点亮指示灯。



计算机的理想模型

模型中各个元素都有不同的物理状态，我们分别以二进制值来表示。这些状态包括指示

灯和开关的开与关、导线是否连接到电源等。状态的变化对应于不同的信息，可以在电路中传递。

为了展示如何与物理世界相连接，在模型中我们采用了二维的几何表示。导线对应于在平面中绘制的线段；控制开关对应以特殊方式交叉的电线；电路就是绘制在矩形框内的若干导线。我们用一个矩形框来表示电路的边界，其中输入端的导线画到边界即终止，输出端的导线需要超出边界。当我们对电路的实现细节不感兴趣时，我们只画出其接口（即矩形框、输入线、输出线和描述标签），如下图所示。

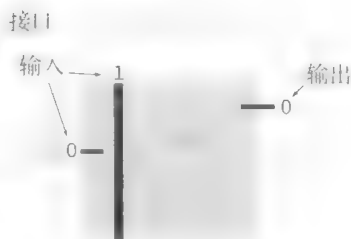
电路其实是一种抽象的表示，不局限于任何一种简单的绘图方法。我们使用这种规定的几何表示法失去了一些灵活性，但是这个方案确实提供了一些方便。这样我们得到的最直接结果就是可以将电路绘制成一个由内部连接线路构成的图纸，以此来表示每个电路的具体实现。如今电路可以依据图纸进行制造，所以设计师和制造商之间对电路表示的一致性就显得非常重要。

我们需要精准地描述关于电路怎样组合在一起的细节。尽管从抽象的角度来看，这些细节并不重要，但从实践的角度来看，我们需要遵守一些简单的规定，以确保我们可以详细地描述电路。这些规定与设计现代处理器核心集成电路所要遵循的“设计规则”几乎没有什么差别。当你看过一些电路后，你不妨重读这段以确保你正确理解了它的基本定义。

**导线** 电路由相互连接的、连向电源以及连向控制开关的导线组成。控制开关是两条导线的交叉点，其中一条导线在交叉之后不久就到了终点。我们将在下一页详细讨论控制开关的操作。在电路图中，我们将导线表示为线段，通常是水平的或竖直的（也有时候是对角线）。它们可以交叉（彼此跨越）或连接（彼此相连）。我们假设每条线总是处于两种状态之一（连向电源或连向地），因此我们可以使用二进制值表示每条线（1 或 0）。互相连接的导线必然具有相同的值。为了便于查看电路中的导线值，我们用粗线代表值为 1 的导线，用细线代表值为 0 的导线。

**连接电源。**为了减少图纸中的混乱，我们假设存在一个始终为 1 的输入，并用一个电源点（power dot）表示电路中任意位置的一个到该输入的连接。在实际的集成电路中，这样的点也可能表示与另外一层上电源的连接。除非连接断开，否则所有连接到电源节点的导线值为 1。任何与值为 1 的导线相连接的导线值也为 1。在后文中，我们将讲述控制开关如何断开连接，并使在此交汇的电线的值翻转至 0。如果输入端口的值为 0，则连接到它的导线值也为 0，同样，与值为 0 的导线连接的导线值为 0。

**输入。**计算机上的输入设备用于向电路提供一组离散的输入值。例如，当你按下键盘上的一个按键，那么对应于此键的统一编码 (Unicode) 值将被输入给计算机的 CPU；如果你滑动屏幕或触摸板，那么你手指的相对移动量对应的二进制数值将被输入给计算机的 CPU；TOY 的开关状态变化直接对应于二进制的输入；等等。输入值的变化将会导致导线和电路内部开关状态的变化，并最终改变输出值。为简单起见，我们假设电路内的状态变化要比外部变化得快——因此，当你按下计算机键盘上的某个键或扭动 TOY 的开关时，计算机可以立即响应。



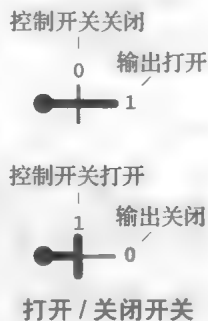
**输出。**输出线的值可以被电路外部的器件或仪器所感知。例如，如果计算机输出一系列字符值的统一编码 (Unicode)，它可能会使得这些字符被连接的打印机打印出来；如果计算机输出一组坐标和颜色值，它会使得显示屏某个位置上出现某种颜色的点；TOY 机的指示灯直接对应二进制输出；等等。通常，电路的输出还可以作为其他电路的输入。

**规定。**为了简单起见，我们的模型通过判断导线是输入端或输出端来判断信息是输入到电路还是从电路输出。在几何表示中，输入端和输出端都位于电路的边界处，从这里它们连接到外部设备，由外部设备来响应电路的开 / 关状态值。这个规定非常容易理解。在我们的图纸中，连接到电路边界的导线是输入；延伸穿过电路边界的导线是输出。输入线表示了电路外部产生的一组二进制值，在流入电路后进行处理；输出线是经过电路计算的一组二进制值，并可以用于在电路外部进行存储或显示等一系列后续处理。一般来说，我们的惯例是将输入端放在电路的顶部或左侧边界，而输出端则是置于底部或右边界。这个惯例是为了方便理解电线的用途。在一些情形中，输入端导线和其他导线或许会沿着水平或竖直方向直接穿过电路，不与其他导线连接。

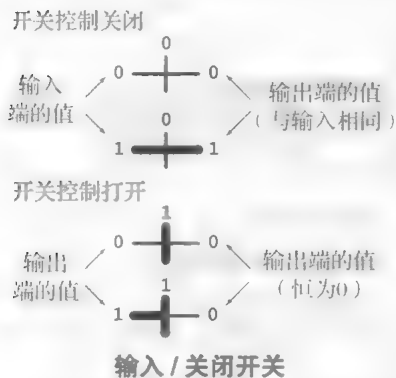
1004

**控制开关** 理解电路的关键在于理解控制开关的操作，即电路中开关控制线 (switch control line) 穿过另一条导线然后结束的位置。开关控制线值的变化可能会断开与其交叉的电线与电源的连接，从而改变该导线的值，过程如下所述。

**打开 / 关闭开关。**在大多数情况下，开关一端的导线直接与值为 1 的电源相连。如果这个开关控制线的值为 0，那么这个开关对导线的状态不会有影响。相反，如果控制线的值为 1，那么开关将会切断连接，使得导线另一端的值变为 0。导线的另一端就是开关的输出。如果开关控制线的值是 0，那么输出端的值是 1；如果开关控制线的值是 1，那么输出端的值是 0。



**输入 / 关闭开关。**更一般来说，我们认为控制开关具有输入值 (不一定是 1)。输入线和输出线连接成直线，开关处于中间位置的交叉点上。



逻辑上，开关的操作很简单：如果开关控制线为 0，则输入线和输出线相连接，因此它们具有相同的值 (均为 0 或均为 1)；如果开关控制线为 1，则输入线和输出线不连接，因此输出端电线的值为 0 (与输入端的值无关)。

也就是说，我们认为控制开关是可以用于切断从输入端到输出端之间的连接控制方式 (通过打开开关控制线)。如我们将看到的那样，控制连接的这种简单的功能可以作为复杂电路的基础，并且是构建计算机电路和其他电子元件电路的关键。

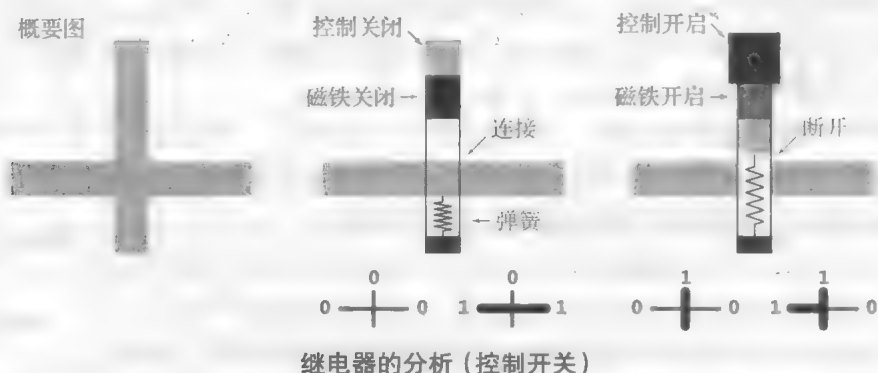
**布局规定。**控制开关的输入是电路的一个输入或另一个开关的输出。控制开关的输出可能用作另一个开关的控制线，也可能是另一个开关的输入端，抑或是一个电路输出。绘制控制开关时，我们并没有明确区分输入和输出。在我们的电路中这种区别是清楚的，因为输入线总是连接到电源或另一个开关的输出线，并且因为输入通常位于我们电路的左侧或顶部，而输出位于右侧或者底部。

1005

**一个实例。**如何构建一个控制开关？为了更加直观，我们分析继电器 (relay) 设备。在继电器中，控制线连接到一块电磁铁，电磁铁通电时可以吸引一小段导线，这段导线能够将

输入线连接到输出线；同时这段导线也连接到弹簧。如果磁铁断开，弹簧将拉动导线使得输入端连接到输出端；如果磁铁接通，它施加比弹簧更强的力来拉动导线，以便断开输入和输出之间的连接。这种继电器仍然运用于各种物理设备中，如烤面包机、收音机、蜂鸣器、门铃等。

我们不用继电器来建造现代计算机是因为它们的体积大、运行慢、成本高，并且连接几百万至几十亿的继电器是很低效的。现在大多数计算机中的开关是通过一种被称为晶体管的微小器件实现的。一些晶体管非常小，甚至比我们绘制图纸中的电线还要小。早期的计算机是由其他类型的开关制成的，包括真空管、继电器和其他种类的设备等。几乎从计算机发明以来，构建效率高、体积小、成本低的计算机在很大程度上取决于构建更快、更小、更高效的开关。



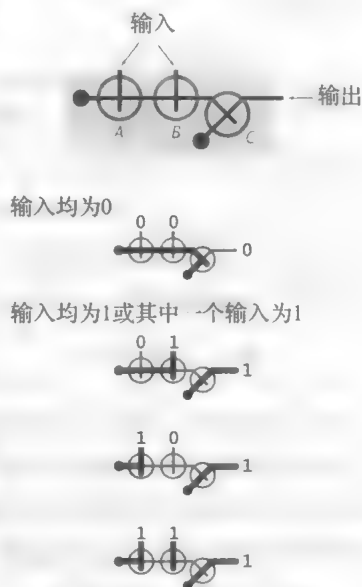
**电路** 在构建电路的过程中，我们可能将电路输入连接到开关控制线，也可能将开关输出连接到其他开关控制线或电路输出。开关控制线值的变化通过我们的电路传输并构成计算。由于这些变化可能很复杂，我们将详细地讲解每个连接。

**开关电路分析。**理解电路的关键在于理解控制开关的操作，以及开关所在的位置对导线的影响。当开关控制线的值改变时，可能导致开关输出的值的变化。如前所述。

举个例子，右图所绘制的电路中有三个开关，分别标有A、B和C。这个电路实现了基本构建要素之一——“或门”，它的输出值是将两个输入值进行逻辑或 (or) 运算，稍后我们会详细描述。要理解这种行为，我们可以分析对于所有可能值的输入情况下开关的响应，如图所示。当输入控制一个开关时，我们可以知道该开关的输出。如果该输出线控制着一个开关，那么我们可以知道该开关的输出，等等。

我们首先来分析两个输入都为0的情况。在这种情况下，A和B都将输入连接到输出，因此开关C的输入为1。然后开关C从其电源点断开连接，因此输出是0。在其他情况下，开关A或开关B断开与左侧电源点的连接，开关C的输入为0，因此不会断开与其电源点的连接，输出为1。这种分析确实有点复杂；幸运的是，我们只需要对为数不多的这种规模的小型电路做分析就可以了。

**组合电路。**虽然定义里并没有明显的体现，但不同类



一个开关电路 (或门) 示意图



型的电路之间存在着一个根本区别，即电路中是否存在环路（loop）。通常，如果电路中没有环路，就像我们前面分析过的电路类型，其最终的输出值总是唯一的。组合电路（combinational circuit）是指没有环路的电路，因此，其输出值仅取决于输入值，与电路的当前状态无关。在组合电路中，一个很经典的例子就是加法器（adder）。它采用  $2n$  个输入值表示两个  $n$  位的二进制值， $n + 1$  位的输出用于表示计算的和。无论以何种顺序输入，我们期望经过短暂的延时后一旦输出值稳定，我们可以看到从电路总是输出相同的求和结果。我们将在 7.3 节中讲述一个完整的加法器电路的设计。

1007

时序电路。相比之下，时序电路（sequential circuit）具有环路。时序电路具有这样的性质：输出值取决于随时间变化的输入序列。计数器（counter）是时序电路的一个经典例子，它可以有一个输入值和  $n$  个输出值。 $n$  个输出是输入从 0 变为 1 并再变回 0 的次数的二进制表示。时序电路的独特之处在于其内部元件的状态不仅取决于输入的当前状态，而且取决于过去的状态。

在你了解一些组合电路或时序电路之后，你将会更好地理解这种区别。组合电路是两种类型电路中较为简单的一种，因此我们将先在 7.3 节中详细介绍组合电路。之后在 7.4 节中对时序电路进行讲解。

正如你所看到的，本节在开始介绍了构建电路中所用到的基础部件，我们使用这些电路来构建几个更大的模块，然后将这些模块的接口组合在一起完成一个处理器。很容易看出，我们从开关到处理器只需要两个层次的抽象。

**逻辑设计和现实世界** 我们的模型侧重于处理器的逻辑设计（logical design），而不是物理实现。该模型并没有考虑机器重量多大、由何种材料组成、需要多少电能、什么颜色，以及其他物理特性。它甚至不要求电路的电气特性。例如，电路中控制开关的设计并不比运输水和煤气的管道更难。我们假设构建的是理想设备，并没有考虑晶体管等新技术。

构建计算机的过程中使用了各种开关，这证明了控制开关这一抽象设计的有效性。事实上，计算性能的提高也建立在更高效的开关的基础上。

在现实世界中，开关和导线类型的差异要仔细考虑，所以我们的抽象只是一个起点。有很多因素并没有充分考虑，如驱动开关所需的功率、开关切换连接所需的时间或开关的物理尺寸等。当我们真正想要制造计算机时就，需要面对所有这些考虑因素以及其他因素了。

1008

在现实中，我们同样需要考虑放置每个导线和开关的位置，为了解决这个问题，首先指定每个电路的布局。你可能会想到，我们开发的每个电路都将对应于具有伸出边缘的导线的物理设备，为了将它们连接在一起，我们需要决定输入和输出电线的物理位置。在现代电路设计中，移动电路中的模块会相对灵活方便些，但每个模块最终都会占据一定的空间。虽然我们并没有完全考虑所有的细节，如导线宽度、导线之间的距离或连线交叉等细节，但我们的方法可以很容易地扩展以支持它们。

导线、电源点、控制开关和电路等概念尽管简单但非常有用。它们足够用来设计任意复杂的机器。它们代表了物理世界与抽象的计算世界之间的狭窄接口，这个抽象世界为我们在技术上改进这些机器提供了无尽的潜力和可能。

1009

## 问答环节

问：开关这个词是指某个抽象的东西，它可以开或关，还是指类似于开灯按钮的真实东西？

答：都是。开关和导线是我们用于设计电路的基本抽象构建块，也是计算机的物理基础。以计算机科学家的观点，开关的制造方式是无关紧要的。电路研究即研究开关网络的集合。我们已经涉及两种不同形式的开关的物理实现：TOY 计算机前面板上的开关和在我们的电路中以交叉线形式实现的控制开关。从科学家或工程师的角度来看，开关至关重要，因为在现实世界中找到新的开关材料或构建新开关可能会带来新的见解或产生新的讨论。开关与现有计算机的已知知识直接相关。开关可能是继电器或晶体管，或物理世界的某些东西，如遗传物质、神经元、分子或黑洞等。

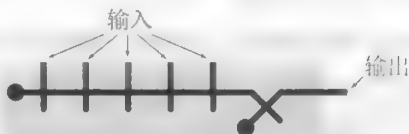
问：电路怎么理解？这个词似乎是指某种循环，但我们主要是指连接在一起的事物。

答：与导线类似，这个术语来自于电子电路，当接通电源时就有了循环。一个灯泡点亮，它必须在这样一个循环；这就是为什么用来连接电源时需要有两个插头。我们的模型并没有解决电路如何工作的问题，你不需要对电气有过多的了解就能理解我们的模型。电路在计算机发展中的作用是不可否认的，对电气的理解自然对计算机的学习也是有帮助的。最基本的要求是你可以理解我们的简单抽象规则，并且对于一组给定的输入行值（物理开关设置），能够确定哪个输出线连接到电源点，从而使得相应的灯点亮。

1010

## 练习

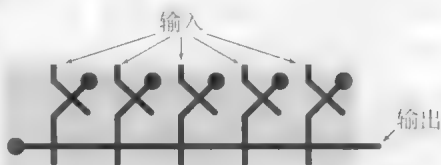
7.2.1 下列电路在什么条件下输出为 0？



答案：当且仅当所有的输入都是 0。

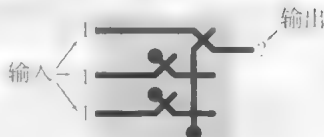
7.2.2 说明练习 7.1.1 在什么条件下输出是 1。

7.2.3 下列电路在什么条件下输出为 1？



7.2.4 说明练习 7.1.3 中输出为 0 的条件。

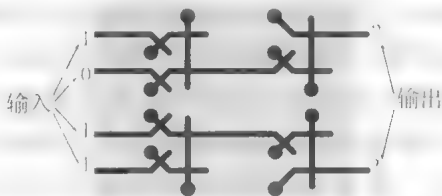
7.2.5 将值为 1 的电线加粗，并给出以下电路的输出。



答案：

注意：在我们的电路中，当输出端电线值为 1 时，其始终连接到一个电源点。

7.2.6 将值为 1 的电线加粗，并给出以下电路的输出。



1011

### 7.3 组合电路

许多计算任务都可以归约为寻找特定数学函数在给定输入情况下的函数值。在本节中，我们重点关注构建电路以完成这部分任务。例如，我们知道任何计算机都需要一个将两个二进制数相加的电路：那么我们如何构建一个这样的电路呢？

解决这个问题的过程中你可能会两个惊喜的发现。第一，有一个简单的系统化的方法来构建电路，它可以计算任何定义明确的布尔函数。我们会在本节中的后续部分介绍这种方法。一旦你掌握了这种方法，你就会发现它仅在输入值较少时适用，因为过多的输入值会导致大量的开关和导线的使用。在设计计算机的过程中，我们不仅要考虑如何为我们需要的函数建立电路，而且还要考虑这样的电路在现实世界是否可行。第二，对于实现 TOY 这样的计算机需要的函数所需的电路，只需要经过一些简单的层次抽象就能开发出来，而且非常容易理解。本节的重点是学习这类电路的原理。

我们会具体讲解如何为布尔函数构建对应的电路。这些电路被称为组合电路 (combinational circuit)，因为它们的输出值取决于它们的输入值。组合电路是理解计算机操作必不可少的出发点。

我们从大家所知的门级电路开始讲解。它们的基本功能是实现“非”“或非”“或”以及“与”等布尔函数。然后，我们将进入更深的层次：通过将众多门级电路连接在一起，构建实现各种逻辑开关的电路。之后，我们考虑将布尔函数的真值表定义转换为实现它的电路的一般结构。为了解释这些模块的构建过程，我们首先从将两个二进制数字相加的电路开始。通过本节内容，我们会对电路组合、电路连接，以及利用它们在更深层次上构建电路等有更深入的了解。

1012

门 正如我们在 7.1 节看到的那样，布尔逻辑是一个抽象的数学系统，在近两个世纪里，它在数学推理中发挥了核心作用。但是它与构建计算机有什么关系呢？这种联系是由克劳德·香农 (Claude Shannon) 在 20 世纪 30 年代在麻省理工学院求学时建立的，这是一项深远的举措。香农认为，我们可以构建实现布尔功能所对应的数字电路 (digital circuit)，因此可以将布尔逻辑直接映射成对应的计算设备。当时一部分人 (包括香农的导师) 正在尝试使用模拟电路 (analog circuit) 构建计算机，这种电路用导线上的电压来表示实数，也有一部分人在没有布尔逻辑提供的严密基础上试图研究数字电路。

下面我们跟随香农用电线和开关来构建称为门的小型设备，从而将电路模型提高到一个更高的抽象层 (自从他的作品出版以来，每个人都已经做过了)。具体来说，我们会构建实现“与”“或”“或非”和“非”的布尔函数的门电路，包括多种输入的情况。正如我们将要看到的，门是非常简单的，用几个开



已获诺基亚公司授权。

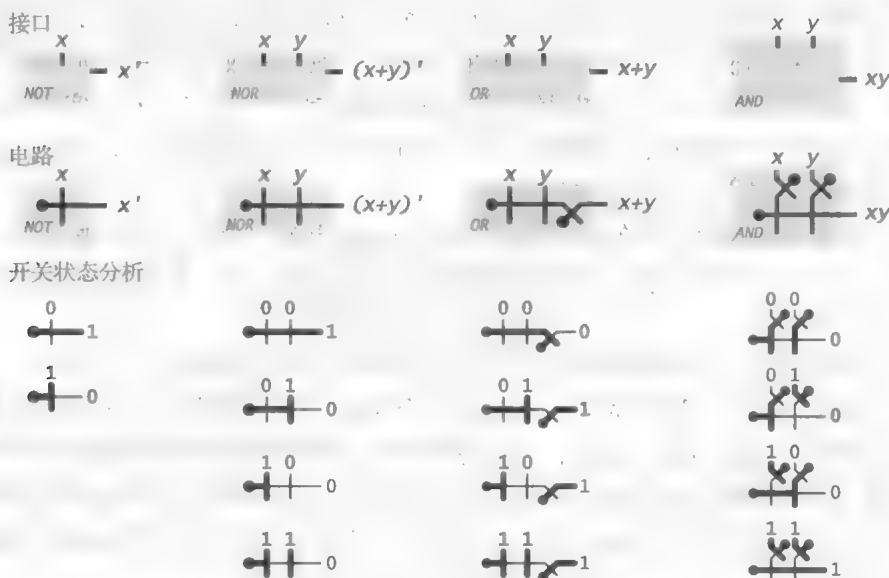
克劳德·香农 (1916—2001)

关就可以构建，并且它的功能也足够强大，可以用来构建任何计算设备。

同所有的电路一样，我们用一个方框表示门电路，其中包含输入线（连接到方框的边缘）、输出线（穿过方框的边缘）和开关，以及将它们连接在一起的导线。门电路的连接都非常简单，其中输入由开关控制，其他连接很简单，如果输入不改变值，输出也不会改变。但是，当输入发生变化时，输出会在很短的时间后发生变化，这段时间称为切换时间（switching time）。换句话说，门级电路是一个简单的电路，每当输入线上的值发生变化，经过一定的时间间隔后，它的输出线上的值是输入线值的基本布尔函数值。

非门。你可能已经注意到，如果我们将开关的控制端视为输入，则我们的“打开 / 关闭”开关就是一个实现布尔“非”功能的门。将开关控制端看作一个输入的话，如果输入为 0，输出为 1；如果输入为 1，则输出为 0。因此，如果输入值是  $x$ ，则输出值是  $x'$ 。因此从现在开始，我们将把“打开 / 关闭”开关称为非门。在下面图中的左侧展示了非门的实现细节。

**1013** 非门有时被称为逆变器（inverter）。



由开关构建的非门、或非门、或门和与门

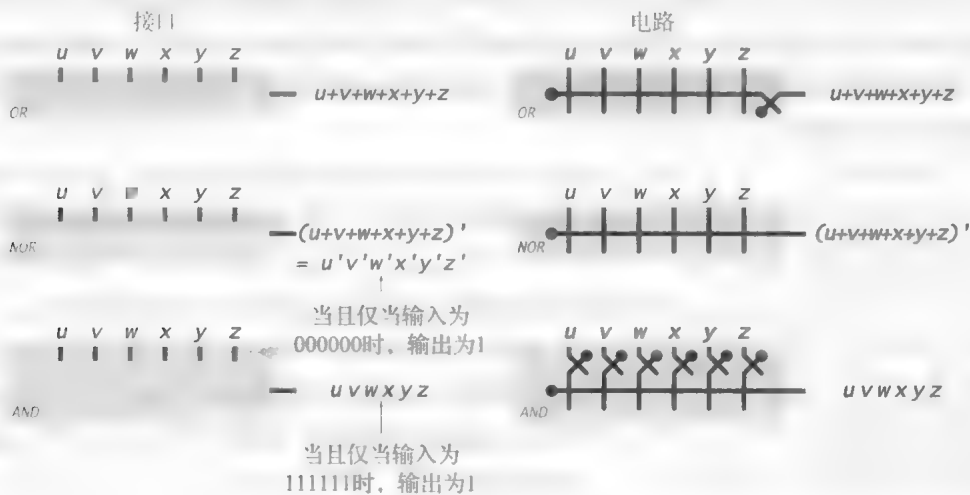
或非门。如上图中第二列所示，将打开 / 关闭开关的输出与输入 / 关闭开关的输入连接可以构建一个门，当且仅当两个输入（开关控制线）都为 0 时，门的输出为 1。当且仅当两个输入都关闭时，输出线才能连接到 1，因为只要任一输入是 1，则连接将被中断。检查真值表，我们看到这个电路实现了布尔或非函数。

或门。“或”的运算即是对“或非”的布尔值进行取反运算，所以我们通过将一个“或非门”的输出连接到一个“非门”来建立一个“或门”。当且仅当两个输入都是 0 时，输出才是 0。你可以查看图中第三列的开关操作的细节来验证真值表，上面这个简单的分析会让你更容易理解。

与门。德·摩根定律为我们提供了一个利用“或非门”打造“与门”的途径。我们对两个输入的结果都取反，即如果  $x$  和  $y$  是输入，那么有  $(x' + y')' = xy$ 。当且仅当两个输入都是 1 时，输出为 1。接下来，你可以通过检查图中第四列的开关操作来验证这一事实，但使用布尔代数的证明过程更容易理解。

**1014**

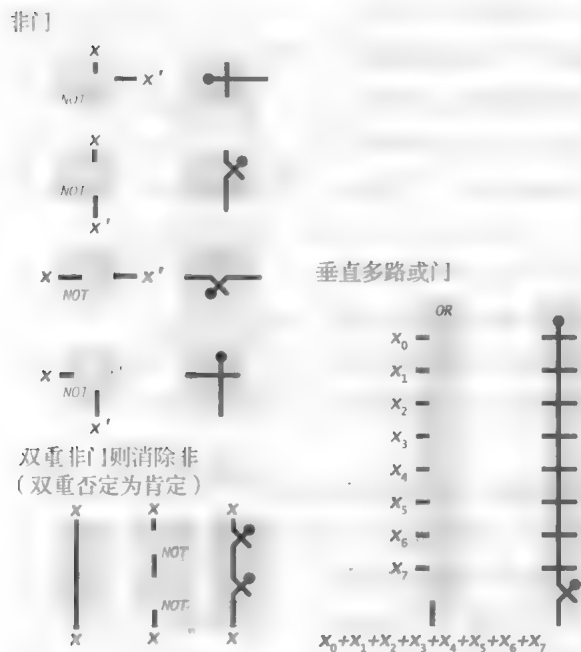
多路门。我们用来构建或非门、或门和与门的方法，可以很自然地进行扩展，以用来处理任意多路输入的情况。为了建立一个  $n$  个输入的或非门，我们把  $n$  个开关（一个“打开/关闭”开关连接  $n-1$  个“输入/关闭”开关）连接起来：最后的输出值是所有输入值求“或非”值。和双路输入门一样，我们通过对或非门的输出进行取反得到或门，对或非门的输入进行取反得到与门。如下所示：



由开关构建多路与、或非、与门

请注意，按照扩展的德·摩根定律（参见练习 7.1.14），可以有两种方法来使用或非门的输出。

几何表示法，旋转，翻转和双重否定。在这一小节中，简单地回顾一下与我们在电路设计中有关的几何图形使用的惯例。一方面，我们总是希望在进行电路设计时，将电路一层一层抽象出来。例如，当我们使用门电路构建更高级别的电路组件时，我们只展示门电路的接口而非电路实现细节。另一方面，我们想让你更直接地感受到电路设计的全过程，因此即使在高度抽象的情况下，我们也会向你展示底层电路设计。这种设计方法产生的一个结果就是：几何图形的使用将是我们所有设计中一个重要考虑因素，即使有时并不必要。例如，在电路中表示与门、或门和非门时，传统电路设计的方法是使用大小相同的不同形状表示不同的门，而我们会使用尺寸略微不同的矩形。



等价电路的例子（左侧：接口；右侧：电路）

口而非电路实现细节。另一方面，我们想让你更直接地感受到电路设计的全过程，因此即使在高度抽象的情况下，我们也会向你展示底层电路设计。这种设计方法产生的一个结果就是：几何图形的使用将是我们所有设计中一个重要考虑因素，即使有时并不必要。例如，在电路中表示与门、或门和非门时，传统电路设计的方法是使用大小相同的不同形状表示不同的门，而我们会使用尺寸略微不同的矩形。

这种方法的一个结果是，在构建更复杂的电路时，有时需要旋转、反射或拉伸我们的门电路，有时会使得这些电路不那么容易理解。左图中显示了一些例子。第一组示例显示了我们传统使用的非门的四种实现方式：输入可以在顶部或左边，输出可以在右边或底部。第二个例子显示，将一个非门的

输出连接到另一个非门的输入相当于什么都没做。这个例子强调了我们不使用输入和输出之间的物理连接,而是逻辑连接。在双非门中,输出线的值等于输入线的值,但它们在物理上并没有直接相连接。最下面的例子显示了一个多路或门,经过反射和旋转,变成垂直摆放的形状。这种类型的门在我们的电路中经常出现,用以收集来自多个资源的值。

当然,将输入/输出从顶部/右侧移动到左侧/底部、消除双非门、对门电路进行反射和旋转,对于给定输入产生的输出值没有任何影响。我们现在展示这些变体以避免在更大的电路中看到它们时产生疑惑。即使是将电路进行  $180^\circ$  旋转或者上下颠倒,都不会影响电路的操作。我们总是希望输入放在左侧或顶部,输出在右侧或底部,这样做的唯一原因是让你更容易看懂电路在做什么。

接口



—  $uv'w'xy'z$

当且仅当输入为  
100101时,输出为1

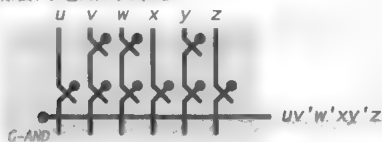
门的实现



—  $uv'w'xy'z$

AND

底层门电路的实现



双重否定消除的电路



广义与门 (一个例子)

广义多路门。扩展多路门的思想,我们还可以选择将某一些输入取反,用以计算如  $uv'w'xy'z$  函数,从而得到一个扩展的广义与门,这些门对于实现组合电路以及底层电路分析是很有必要的,也值得研究。

首先,请注意前文的多路“与门”具有一个重要的属性,即只有一组输入值可以使得输出为1。我们可以利用这个属性,将其中一些输入取反,从而计算若干个小项组成的“与”函数。

左边最上方的图展示了我们用于这种门电路的接口。它描绘了计算函数  $uv'w'xy'z$  的“广义与门”的接口。规则很简单:当输入线上有一个圆时,对输入取反。第二张图展示了如何使用非门和与门来实现这样的电路。而在第三张图显示的电路中,任何取反的输入都会对应于一个双非门,所以所有这些非门可以被删除,留下简化后的电路,如最下面的图所示。如果你不确定这个电路是如何工作的,可以参考右侧图,这是一个开关状态分析,展示了对于三个输入的“广义与门”的所有可能组合,并计算了函数  $uv'w'$ 。

这些门电路简洁且灵活,它们构成了我们将在本节稍后讲解的电路的基础。如果你还不能理解它们是如何工作的,你可以稍后复习一遍。但是要想知道的是,其实从门电路的接口处就能很容易地看出来它的功能——如果导线表示1,圆圈表示0,那么接口表示的二进制数就是使得输出是1的输入值。

比起开关,门电路为我们提供了更高的抽象层次。它给我们的最大贡献是使我们能够忽略使用开关工作的细节,而使用布尔逻辑来思考电路设计问题。为清楚和方便起见,我们已经考虑了非门、或非门、与门和或门的明确构造,并扩展了具有多个输入的情况。在实际使用中你可以看到,考虑到“非”是一个单输入

接口

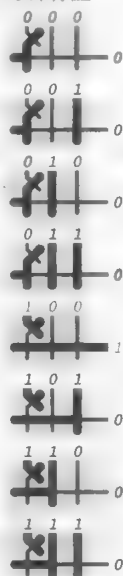


AND

—  $uv'w'$

当且仅当输入为  
100时,输出为1

输入的所有值



广义与门的开关分析

的“或非”，我们可以只用或非门来构建非门。

我们使用控制开关构建门实际上是对开关非常有限的应用：每个开关要么是被用作逆变器，要么就是用于控制 1 和输出线之间的切换。我们可以用更少的开关建立门，但是我们这样的设计可能在现实世界中更好实现，因为必须考虑到我们会同时建造数百万或数十亿的开关，而如何逐一控制它们是个艰巨的任务。例如，每个门的输出线都有一个到电源的连接。

通常，我们会想在使用更高层次的抽象时我们丢失了什么信息。实际上这是无法知道的：因为控制开关确实可以以各种方式相互连接，而且它们组成的电路的行为也难以理解。然而，通过布尔逻辑我们可以证明，任何使用门电路来构建的电路都可以使用控制开关电路来搭建，反之亦然。

同样地，我们也总是会考虑在使用更高层次的抽象时获得了什么。在这种情况下，电路分析会变得更加容易和更系统化，但更重要的是，这个抽象层次是深刻的，因为它把物质世界与抽象世界分开了。

向底层硬件的方向看，门电路的使用使我们免于担心特定的复杂物理设备的行为。也许我们可以使用不同类型的弹簧、磁力更大的磁铁，等等；我们甚至可以不使用开关来构建门电路。这种在底层完全改变物理实现的能力是推动计算进步的根本力量，几乎从它诞生之初就开始了。门电路已经经历过使用继电器、真空管、晶体管以及其他许多物理方法来实现。在这个过程中我们看到，改善一切的一个办法就是做一个更好的开关。实际上，改善一切的另一种方法是建立一个更好的或非门（参见练习 7.2.2）。

往上层电路设计方向看，门电路的使用将我们的电路与布尔代数联系起来，布尔代数是一种完全成熟的数学系统，我们可以利用它来进行开发。这是香农的见解。关于布尔函数性质的缜密数学表述也帮助我们构建起理解数字电路行为的基础。

1018

每一个门本身就是一个电路，所以我们可以简明地为我们的电路抽象出一个递归定义：

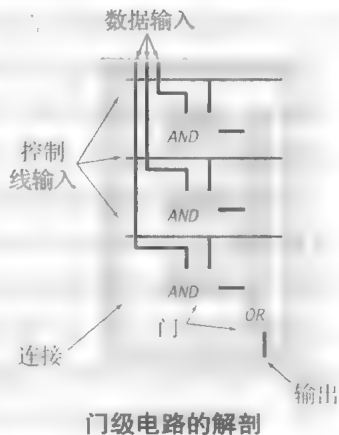
**电路**即通过导线将门电路或者小部分导线网络连接而成，其中一些导线会被作输入端和输出端。

尽管它已经很简单了，但为了清晰起见，我们将稍微改进这个定义，因为我们接下来将在几个不同的抽象层次上构建越来越复杂的电路。

**用门构建电路** 从编写 Java 程序的经验来看，通过定义合适的接口并遵循相应的规定，你已经充分理解了将小程序构建成大程序的强大功能。同样的想法贯穿于硬件设计中。从现在开始，我们将通过将门电路连接在一起开发更复杂的电路。我们将遵循以下约定：

- 如前所述，从电路顶部和左侧输入，自电路右侧或底部输出。
- 一些输入线被称为控制线（control line），我们将它们标记为粗体的，它们可能以多种方式贯穿在电路中。
- 门电路上会标记它的功能，并按照接口的使用方法进行连接。
- 所有不在表层直接使用的开关电路都被称为门。

例如，右边的电路由三个与门和一个或门组成，具有三个输入、三条控制线和一个输出。区分控制数据移动的输入线和传输数据的输入线是很有必要的。为了方便起见，我们允许控制线遍布整个电路，以便于我们可以在任何一处都可以与它们





连接。这有时违反了输入应该来自左边的规定，但是实际上这并不影响对电路的理解，因为我们的约定实际的意义是要求数据总是从左侧流向右侧，而控制线上的并不是数据。

例子：选择多路复用器。 $k$ 路选择多路复用器或简称为 $k$ 路多路复用器( $k$ -way mux)是具有 $k$ 组输入线和一条输出线的组合电路，能够将其中一个输入值转移到输出，如下所示：每組输入包括数据输入(data input)线和控制线(control line)。其中控制线至多有一个为1，用于选通相应的输入到输出的线路连接。也就是说，电路其实在实现一个逻辑开关(logical switch)的功能，让输出值与所选的输入值相等。我们强调这是个“逻辑”上的开关，因为我们不希望导线在物理上连接，只是有指定的值。

右图中顶部的接口是3路选择多路复用器的示例。输入值标记为 $x$ 、 $y$ 和 $z$ ，控制线标记为 $s_x$ 、 $s_y$ 和 $s_z$ 。如果 $s_x$ 是1，则电路将输出值设置为 $x$ 的值，以此类推。接口还指定了电路的大小和形状，以及输入和输出的位置。像往常一样，这些几何约束对于理解电路的功能并不重要，但是坚持这些几何约束对于理解底层电路以及将其连接到其他电路时会更容易。

中间的图具体描述了如何实现这一功能。实际上，电路实现的是布尔函数 $s_x x + s_y y + s_z z$ 的功能(这个图并没有控制在至多只有一个选择线是0，解决方案详见练习7.3.4)。

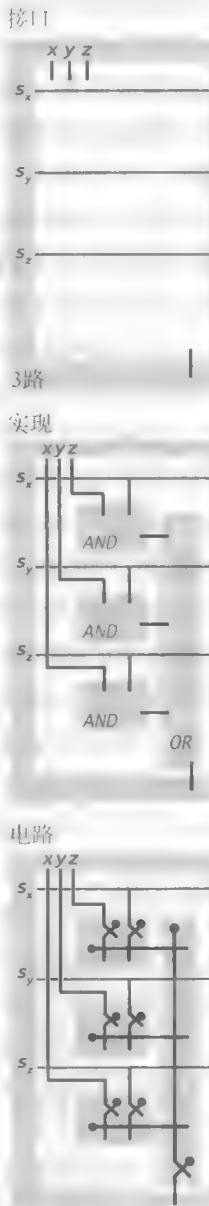
最下面的图比较直观，我们可以看到下面隐藏的门级电路是如何实现的。我们经常会这样做以使我们能够理清整个电路的各种属性。例如，你可以看到在这个电路中，输入和输出之间没有任何物理连接。

你在后面的内容中会看到，三路选择复用器不仅仅是一个玩具一样的简单示例，我们会使用这样的电路来切换处理器组件之间的逻辑连接。更直接地，你可以把多路选择复用器想象成一种切换机制，就像把多种信号连接到你的电视机或计算机显示器，然后用一个切换器来选择其中一种信号进行显示(实际上，这样的多路信号选择器真的存在，而且价格不贵)。

门电路设计以及前面提到的简单规定，大大简化了构建和理解电路的过程。大多数人从门电路开始学习电路设计。与软件一样，你很快就会习惯于在更高的抽象层次上工作。

我们的方法要求在整个过程遵循几何约束，从而略微增加了复杂性，但这些额外的工作产生了相当多的好处，使得整个电路都显得更加有条理。

**解码器、多路分配器和多路复用器** 为了更好地解释门电路用于电路设计的过程，我们在接下来的例子中展示几个完全由与门构建的组合电路(或许会用到一个或门)。这会是你学习利用门电路搭建复杂电路的一个良好开端。万事开头难，但这个过程其实很容易理解。这三种电路都包含 $n$ 条自上而下的输入线，内部有 $2^n$ 个 $n$ 输入与门，每一个门都可以按照逻辑实现对输入值的传递或者是取反后传递。在实际使用中，这些电路也会因为使用场景的不同、要求的输入或输出信号的不同产生略微的变化。

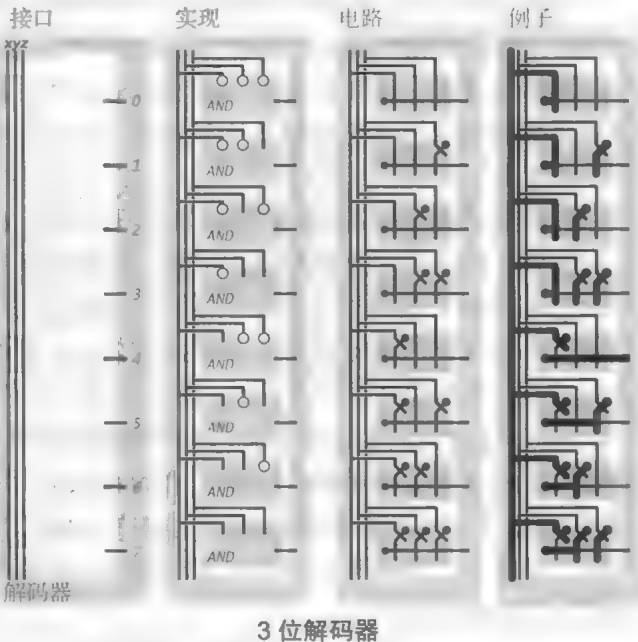


一个3路选择多路复用器

**解码器。**解码器是一种组合电路，一个具有  $n$  条输入线的解码器有  $2^n$  条输出线。如果我们将输入看作一串二进制数字，并对所有的输出线进行编号，那么解码器的作用就是选中输入信号所指定的输出线。具体而言，将  $n$  个输入值解释为  $n$  位二进制数  $i$ ，并且将输出线编号为  $0$  至  $2^n-1$ ，解码器中所选中的第  $i$  条输出线的值为  $1$ ，其他输出线的值则为  $0$ 。如右图所示，最左边的接口图中画出了  $3$  位解码器的大小和形状，以及其输入和输出的位置。

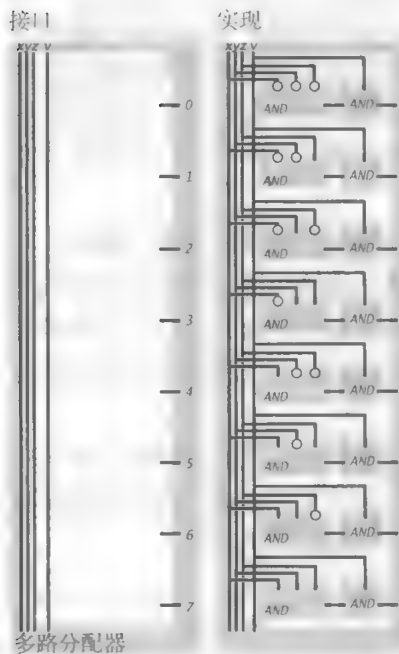
左数第二个图展示的是如何使用  $2^n$  个与门搭建  $n$  个输入的解码器。对于每个输出线，仅有其相对的一组输入值可以使其输出值为  $1$ 。从图中可以看到，在某个输出线输出为  $1$  时，如果需要相应的输入线输入为  $0$ ，则对应的输入线上有取反标记；如果需要输入线输入为  $1$ ，则直接连接进与门中。左数第三个图展示的是具体的电路实现。在最右侧的图中，我们对其中的开关进行状态分析，如电路将输入值  $100$  解释为二进制数  $4$ ，并将输出线  $4$  的值设置为  $1$ （所有其他输出为  $0$ ）。

你将在  $7.5$  节中看到，解码器电路在我们的处理器中扮演着关键的角色，我们可以将计算机指令中的二进制代码转换为对应的某根输出线值为  $1$ ，从而激活由二进制码寻址的电路。



3 位解码器

1021



3 位多路分配器

在图中同时还列出了开关层级的分析，你可以看到每个与门是如何响应给定的一组输入的。在接下来，我们将不再展示这些电路内部的原理图，因为我们从门级电路中可以很清楚地看见电路的实现逻辑。

**多路分配器。**如左侧图所示，一个多路分配器 (demultiplexer, 简称 demux) 是在解码器的基础上再增加一个输入信号 (我们称之为输入值)，以此形成一个  $1$  对  $2n$  的逻辑开关，将输入值切换到解码器选定的输出线上。换句话说，除了由地址输入的二进制值选中的输出线以外，其他输出线均输出  $0$ ；而对于选中的那个输出线，如果输入值为  $0$ ，则为  $0$ ；如果输入值为  $1$ ，则为  $1$ 。正如你所看到的，我们通过给每个解码器输出添加一个与门来实现这一操作，只有输入值为  $1$  时，多路分配器的输出值才为  $1$ 。简单起见，我们用与门选取左侧输入中的一个值而不是将输入值连接过来，如练习  $7.3.2$  中所述。我们不再画出详细的电路图细节，因为它与前文中的解码器非常相似。

与以前一样, 请注意, 多路分配器是一个逻辑开关——输入和所选输出之间没有物理连接, 我们只是简单地让选定的输出具有与输入线相同的值。

1022

在接下来的 7.5 节我们也会提及, 多路分配器会在我们的处理器中发挥关键作用, 因为它们可以按照计算机指令的控制, 将指令中的二进制数据值传导到电路中的其他部分。

**多路复用器。**多路复用器 (multiplexer, 简称 mux) 是一种组合电路, 具有  $n + 2^n$  条输入线和一条输出线, 它的作用是从  $2^n$  个输入值中选取一个作为输出, 因此可以被看作一个  $2^n$  对 1 的逻辑开关。

右边的图展示了多路复用器的实现, 从图中可以看到, 就像在多路分配器中一样, 在解码器的基础上再增加了一个与门, 将解码器的输出送入与门中, 并将与门的输出用一个多路的或门 (在图中垂直摆放) 来收集。或门的输入中最多只有一个为 1 (当且仅当被寻址的线路的输入为 1 的情况), 因此输出值的逻辑是正确的。例如, 如果输入  $xyz$  是 110, 则输入行 6 上的值将出现在输出上。像往常一样, 这是一个逻辑开关——在选定的输入和输出间没有任何物理连接。

显然你也可以想到, 多路复用器在我们的处理器中也起着至关重要的作用, 因为它们允许我们在计算机指令中使用二进制代码来指定用于输出的数据值。

**垂直多路单热或门。**值得强调的是, 我们在多路复用器中用到的垂直多路“或”门其实非常容易

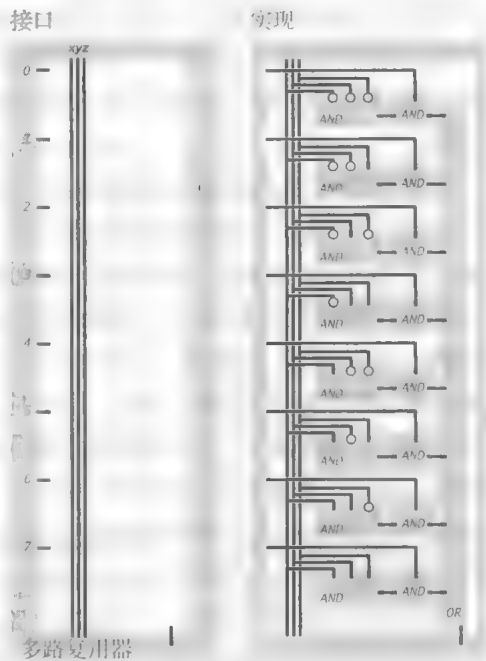
分析, 因为它们的输入始终满足一个特殊条件: 多路或门上的输入仅有 1 路为 1 (热)。我们用“单热”这个名称来强调这个不变的特性。原则上, 利用这个特性我们可以制造很多特殊的电路, 在这里我们使用了一个普通的多路或门。这个或门正好满足我们的需要, 因为广义“与”门中有且仅有一个输出可以为 1——对应于输入数据选中的线。实际上, 在我们的电路设计中, 这是唯一需要使用或门的地方。因此, 当我们提到“或门”的时候, 我们就是指这样一个门。

1023

至此, 我们已经介绍了四个经典的组合电路, 都是由垂直的与门构建的。为了确保你可以更全面地了解它们的工作原理, 我们花了如此大的篇幅来进行讲解。这些例子很好地说明了我们在构建复杂电路时始终遵循的规定, 以及这些基本的电路将在接下来的 7.5 节构建计算机处理器中发挥着关键的作用。更重要的是, 充分了解这些电路的设计将会更充分地锻炼你对于电路的抽象概括能力。

**积之和电路** 值得注意的是, 通过扩展我们刚刚使用的基本方法, 我们可以将一系列与门放在一起建立一个电路, 其输出值可以是输入值的任何一个指定的布尔函数。我们可以从指定函数的真值表直接构建电路。具体来说, 我们需要完成以下几点:

- 选定函数真值表中为 1 的行。
- 将输入值连接到广义与门, 使得当且仅当输入值为选取的这些行的值时, 与门对应的输出值为 1。

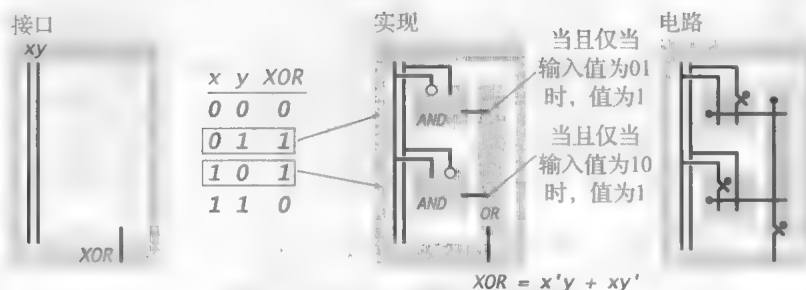


3 位多路复用器

- 将所有这些门的输出送入（垂直）多路“或”门。
- 将该或门的输出作为函数值。

这个结构与我们用于推导布尔函数的积之和表达式结构完全相同。

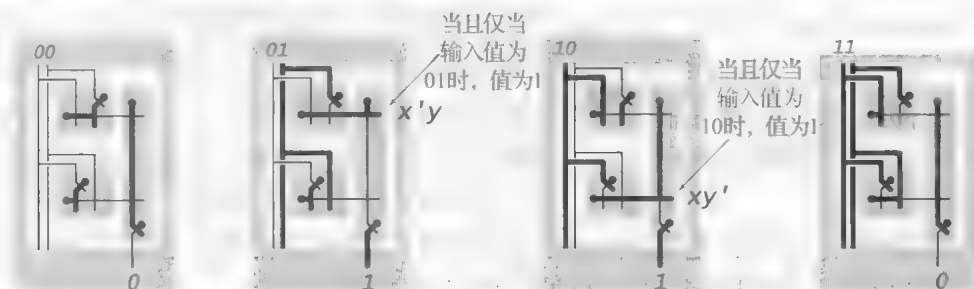
异或。作为第一个示例，我们来分析下面的结构，这是一个由 3 个门电路实现的两个变量的异或函数。



利用真值表构建异或电路

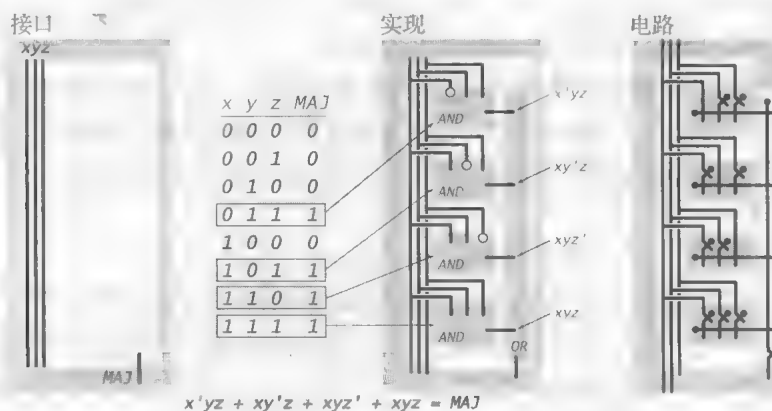
我们首先搭建  $x'y$  和  $xy'$  对应的广义与门，然后将它们的输出送入一个（垂直）或门来计算  $x'y + xy'$ 。如果想验证一下电路工作是否如我们设计所想，你可以参照下图，它会在开关层面为你分析所有输入的可能结果。

1024



稍后我们会利用这个电路实现我们的计算机处理器中的按位异或指令。

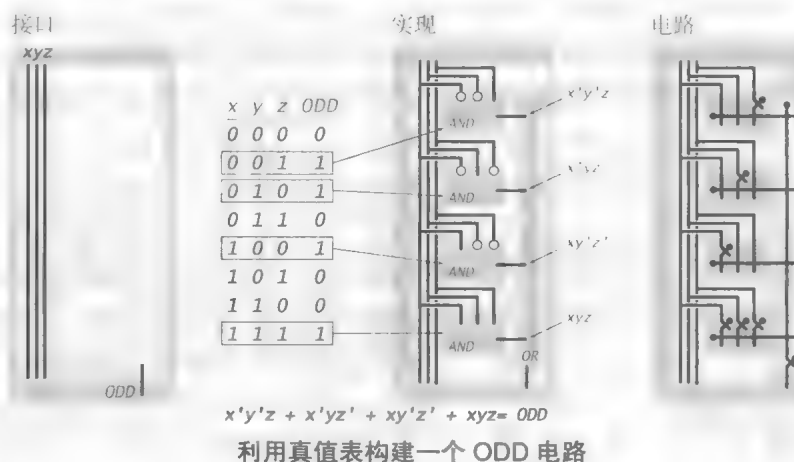
MAJ 和 ODD。同样，为了计算 3 个输入的 MAJ 函数（MAJ 函数又称为表决器，当且仅当多数输入为 1 时输出为 1——译者注），我们可以将计算  $x'yz$ ,  $xy'z$ ,  $xyz'$  和  $xyz$  的门叠在一起，然后对输出进行或运算，如下所示：



利用真值表构建一个 MAJ 电路

当然，我们也可以设计一个电路来计算 3 个输入的 ODD 函数，把计算  $x'y'z$ 、 $x'yz'$ 、

1025  $xy'z'$  和  $xyz$  的与门的输出连接到一个垂直的 OR 门即可：



显然，同样的方法对于任何布尔函数都是有效的。如果你有一个定义清晰的布尔函数，那么你就可以列出它的真值表；一旦你有了真值表，你就可以建立一个电路来计算它指定的函数。有了这个方法，我们可以很容易地构建计算任何布尔函数的电路，这件事情意义重大。

实际操作中的限制。但是，你可能已经注意到了一个问题：积之和电路在计算输入量较大的布尔函数时构建电路将会变得复杂，因为所需门的数量可以是输入数量的指数级。例如，使用这种方法来计算 64 个输入的表决器函数将需要多于  $2^{63}$  个门，显然这是不可行的，而这仅仅是这样一个简单的函数。上一节我们学过的多路分配器电路也有类似的问题：一个  $n$  位多路分配器有  $2^n + n + 1$  个输入，所以如果我们要使用这个结构，我们需要考虑一个  $2^{2^n + n + 1}$  行的真值表。这样的电路显然是不切实际的，例如，我们的 4 位多路分配器只需要 17 个门，但是使用积之和电路设计方法的真值表会有 200 多万行。

但是，积之和电路在实际使用中非常有用。我们可以把电路设计的过程想像成一个盒子，盒子里有多个门电路，每一个对应于真值表的一行，对于任何一个布尔函数，我们只需要将其真值表中为 1 的行对应的门连接起来，就可以设计出相应的计算电路；也可以想象一个大的解码器，只需在真值表中对应于 0 的地方插入非门。再或者，也想像有一个程序会自动地将真值表转换成积之和电路，因为它们只是同一个抽象的不同描述。实际上，这些想像与历史上某些阶段构建计算机的方式并没有太大区别。

1026

一般来说，我们会使用规范的布尔表达式来进行设计，然后再根据相关的定理进行化简，以实现电路设计的优化。通常情况下，这样的优化会找到一种新的电路设计方案，比我们之前使用积之和电路方法所需的门更少。例如，根据下面这个等式（你可以用真值表检查它的正确性）：

$$MAJ(x, y, z) = xy + xz + yz$$

我们可以立即给出一个电路，与我们依据真值表所使用的 4 个三路“与”门方案相比，它只使用了 3 个两路“与”门（见练习 7.2.4）。“电路优化”这一主题在计算科学的初期（那个阶段每个门都还是一个独立的物理实体）进行了大量的研究，因此对于许多常见的布尔函数，科学家已经知道如何使用尽可能少的门来实现。为了讲述清楚，我们将这些简化实现的过程放在本章的练习题中再做分析。

积之和电路给我们呈现了另一个层次的抽象。我们知道如果能够做出  $2^n$  个门，那么我们就能够为任意一个  $n$  个变量的布尔逻辑表达式建立其对应的计算电路。这个结论是我们在上一个抽象的基础上的一个很重要的进步。在上一个层次的抽象中，我们学会了从开关和电线开始搭建电路，并把电路想象成一个黑盒；而这一次，我们找到了一套切实可行的系统方法用于构建具有少量输入的布尔函数的电路，并且我们知道可以为任何布尔函数构建电路。在计算了高度和宽度之后，我们可以为任何布尔函数绘制接口，类似于我们为 MAJ 和 ODD 绘制接口图（当输入数量很大时，我们需要找到一些方法来减少资源使用。）

我们在本节中分析的所有电路都是有用的，但它们执行的计算都相对简单。接下来，我们分析一个电路，它执行我们在日常生活中会遇到的计算任务：将两个  $n$  位数相加。

**加法器** 让我们仔细研究将两个二进制数字相加的过程，使用你在小学时学到的方法。右边是一个计算  $5 + 6 = 11$  的图，用 4 位二进制数表示： $0101 + 0110 = 1011$ 。图的下半部分定义出了任意 4 位二进制加法中各个位的符号名称。

$$\begin{array}{r} 0\ 1\ 0\ 0 \\ 0\ 1\ 0\ 1 \\ \hline 0\ 1\ 1\ 0 \\ 1\ 0\ 1\ 1 \end{array}$$

1027

$c_4\ c_3\ c_2\ c_1\ c_0$  — 进位  
 $x_3\ x_2\ x_1\ x_0$  — 输入位  
 $y_3\ y_2\ y_1\ y_0$  — 输入位  
 $z_3\ z_2\ z_1\ z_0$  — 输出位  
**4 位加法**

通常，4 位加法器电路将具有 8 个输入位  $x_3x_2x_1x_0$  和  $y_3y_2y_1y_0$ 、4 个输出位  $z_3z_2z_1z_0$ ，并且还有 5 个进位  $c_4c_3c_2c_1c_0$ 。可以把  $c_0$  看作一个额外的输入值（我们设为 0），并把  $c_4$  作为一个额外的输出值（我们可以忽略它，或者用来检测是否发生溢出）。

基于积之和电路的实现（草案）。一个  $n$  位加法器可以用  $n + 1$  个组合电路（每个输出位对应一个）来实现，每个电路具有  $2n + 1$  个输入。我们可以考虑用积之和电路来实现，但是，由于真值表有  $2^{2n+1}$  行，所以我们还是放弃了这种方法。存在这样的想法是很好的，但是我们必须找到一个使用合理数量的门的电路实现。

波纹进位加法器（ripple-carry adder）。相反，我们开发一个电路，它采用人类计算两个二进制数字之和的方法。在上面的例子中，首先你将最右一列的 1 和 0 相加（以及隐含的进位，在这里是 0）以得到一个最右边的输出位 1 和一个进位 0；其次右面的第二列 0 和 1 再加上前一位的进位 0，得到输出位 1 和另一个进位 0；然后把这个进位加到从右数第三列的两个输入 1，得到输出  $0^\ominus$  和进位 1；最后把这个进位加到两个最左边的输入 0，得到输出 1 和进位 0。

对于每一位的运算，这个计算过程等价于两个三输入的布尔运算值：计算输出位和下一位的进位。那么，这两个计算的布尔函数是什么？

- 对于进位，如果三个输入位中 1 的个数为 0 或 1，则进位为 0，如果输入位中 1 的个数是 2 或 3，则进位为 1，这是一个三位的表决器函数（majority function）。
- 对于输出位，如果三个输入位中 1 的个数为 1 或 3，则输出为 1，如果输入位中 1 的个数是 0 或 2，则输出位为 0，这是三位的奇偶校验函数（odd parity function）。

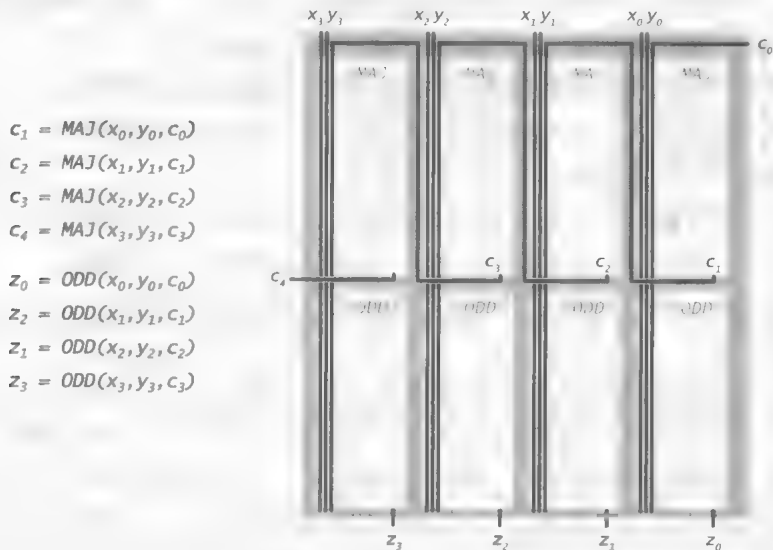
1028

由此，我们可以写出下图左边中的布尔算式，这些算式可以告诉我们如何根据输入值和进位值使用布尔函数运算来得出每一个输出值。根据这些算式，我们就可以构建出一个 4 位加法器的草案图，如算式的右边所展示的那样。我们使用 4 个表决器函数和 4 个奇偶校验函数组件，按照这些算式所示的方式连接在一起。这个结构就是我们所说的波纹进位加法器（又称串行进位加法器——译者注）。从右边开始，首先我们提供输入  $c_0$ 、 $x_0$  和  $y_0$  进入右边的 MAJ 和 ODD 电路，计算进位  $c_1$  和输出  $z_0$ 。然后  $c_1$ 、 $x_1$  和  $y_1$  输入到 MAJ 和 ODD 电路，计

⊖ 原书中此处笔误。——译者注

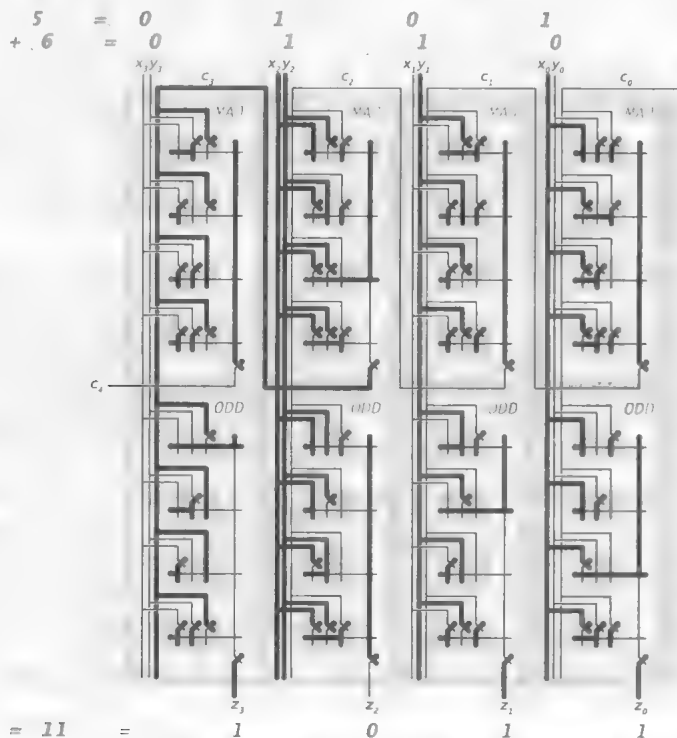
算进位  $c_2$  和输出  $z_1$ , 等等。从右到左依次计算, 像泛起的波纹一样。

现在计算机中已经摒弃了这种波纹式的串行方法而采用了更复杂的加法器实现, 从而在计算较多位数的加法时获得更好的性能。但是我们的电路很好地说明了计算机在执行计算时的基本思路。请注意, 在这里我们在构建电路时又提高了一个层次: 因为这一电路是由 MAJ 和 ODD 电路构建的, 而 MAJ 和 ODD 是由门级电路构建的。



一个 4 位波纹进位加法器

当 4 位加法器的输入被设置为执行计算  $5 + 6 = 11$  时, 下图在开关级分析和展示了底层电路上所有连线上的值, 这非常值得仔细研究。



一个 4 位加法器计算  $5 + 6 = 11$  的开关级电路分析

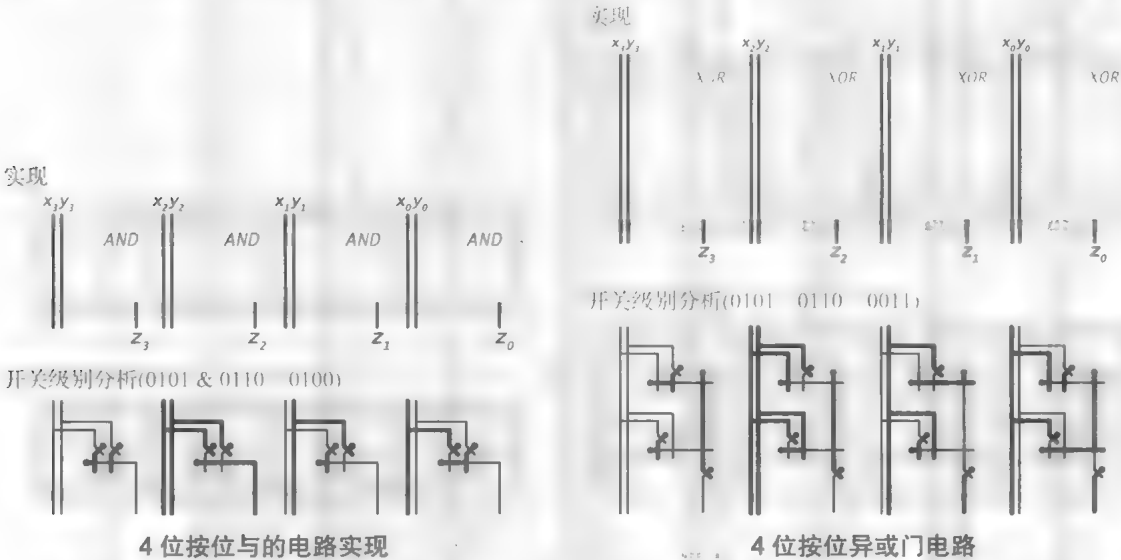


这种结构可以很轻松地扩展到  $n$  位，构建的电路中有  $2n + 1$  条输入线和  $n + 1$  条输出线，可以实现将两个  $n$  位数相加，而只需要使用  $8n$  个通用的 AND 门和  $2n$  个多路 OR 门。也就是说，我们的 4 位加法器有 40 个门，32 位加法器只有 320 个门。当然，这些数字比我们之前所用的  $2^{2n+1}$  行的真值表（这将涉及超过  $9 \times 10^{18}$  个门）的方法要简洁许多。

1030

**算术逻辑单元 (ALU)** 为了给像 TOY 机这样的机器构建一个 ALU，我们需要一个设备来实现这些算术和逻辑指令：加 (add)，减 (subtract)，与 (and)，异或 (exclusive or)，左移 (shift left)，右移 (shift right)。为描述简单并能够突出构建过程的重点，我们将以 TOY-8 为例进行讲解。TOY-8 是第 6 章末尾提到的 TOY 系列计算机的成员之一，在 7.5 节中我们将详细介绍它的实现。对于 ALU 来说，这意味着我们只需要实现加、减、与、异或的功能。我们已经学习了所有我们需要的基本电路的实现方法；现在我们将看到如何把它们整合在一起，组成一个独立的设备，它的输入端为两个  $n$  位二进制值和 3 个控制线，输出端为  $n$  位二进制值（我们忽略加法器的进位输入和进位输出）。控制线的目的是选择所需的计算结果做输出。

**位操作。**按位与 (and) 和异或 (exclusive or) 电路实现起来非常简单，如下图所示。对于“与”，我们简单地用一个与门来表示每一位。对于“异或”，我们使用前面例子中提到的 XOR 积之和电路，每一位对应一个这样的电路。



**输入。**我们的加法器、按位与和按位异或电路都采用了相同的输入。从三个电路的接口图中可以看到，它们的输入数据从左侧穿过整个电路，这使得它们可以左侧对齐堆叠在一起。我们对所有电路都使用了这样的设计，以方便将相同的输入数据传输到每个电路。

1031

**输出。**三个电路的输出都是  $n$  个值（加法器还有进位），但我们只会选择一组值作为 ALU 一次计算的输出。为此，我们只须将三个输出的每一个对应位连接到一个三路的多路选择复用器（我们讲过的第一个门级电路），并将控制线同时接通这些多路选择复用器，使得选定操作的计算结果成为多路复用器的输出。

将这些部分放在一起，就可以为 TOY-8 绘制一个完整的 8 位 ALU，如后图所示。它实现了加法、按位与、按位异或的操作，有 16 路输入，并生成 8 路输出，具体的计算功能由 3 条控制线指定。当控制线中仅有一个是 1 时，输出就是被选中的电路的计算输出。有趣的是，不管哪个计算电路被控制线选中，其他电路结果也是计算完成了的，只是被忽略而已。由此得出：你的计算机正在计算大量完全被忽略的结果！

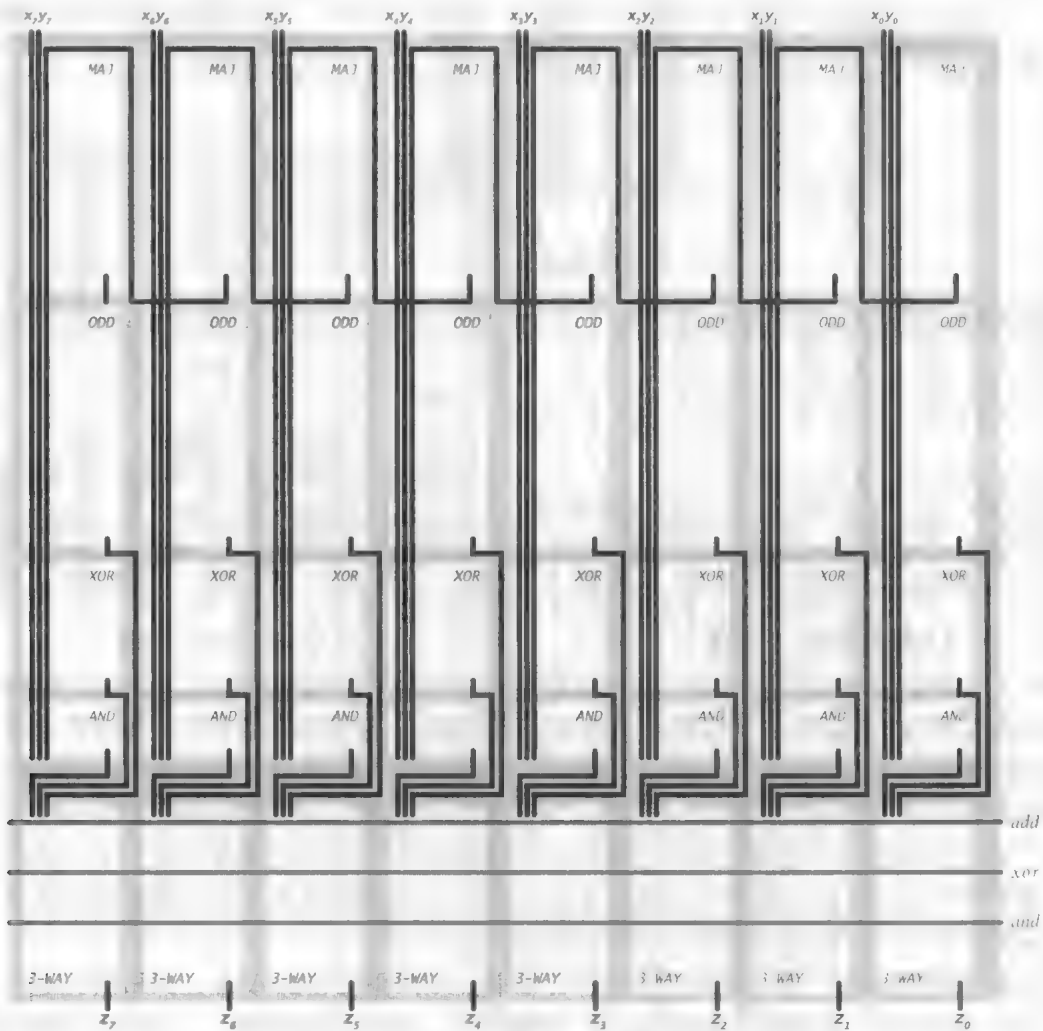
如果将门级电路的实现都揭开，就能看到所有的细节，而 ALU 本身在这个抽象层次上最好理解。同时，你也可以看到每一个开关在计算中发挥的作用。但这里我们不再详述，在本章结尾我们设计一个完整的处理器时，你将看到所有的开关。

这个电路本质上与你的计算机中的 ALU 基本相同，因此通过仔细研究，你可以了解你的计算机是如何执行算术运算的。你的计算机可能有更多的组件，一次能够计算更多的位，但是你也会看到如何在这两个维度上扩展我们的设计。右表给出了该设计用于  $n$  位 ALU 所需要的门级电路的数量。

|       |       | 4  | 8   | 64   |
|-------|-------|----|-----|------|
| 加法器   | $10n$ | 40 | 80  | 640  |
| 异或    | $3n$  | 12 | 24  | 192  |
| 与     | $n$   | 4  | 8   | 64   |
| 3路复用器 | $4n$  | 16 | 32  | 256  |
| 总数    | $18n$ | 72 | 144 | 1152 |

这个 ALU 的设计是一个激动人心的过程，也是一个  $n$  位算术逻辑单元 (ALU) 中门的数量是对我们抽象能力的证明，更是我们对组合电路研究的一个总结。ALU 模块在任何计算机处理器中都扮演着极其重要的角色，我们将在 7.5 节讲解 TOY-8 电路的过程中再着重分析它的实现过程。

1032



一个 8 位算术逻辑单元

1033

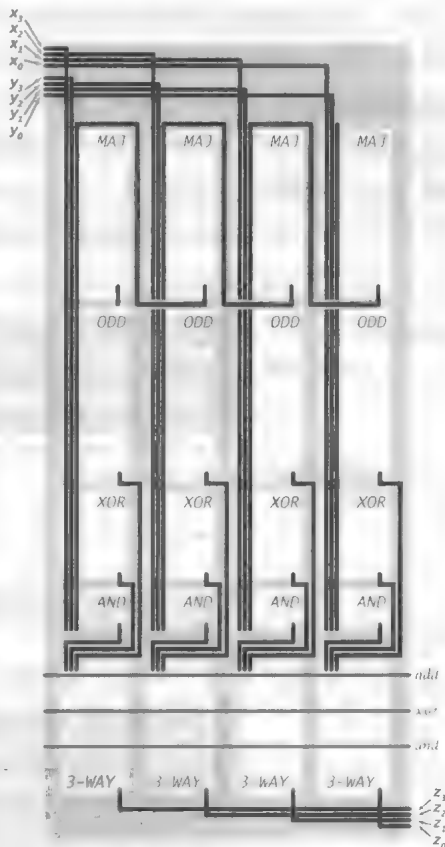
**模块和总线** 我们已经看到，组合电路让计算机拥有了计算布尔函数的能力，因此，它们在“微观”级别的计算电路中扮演着重要的角色。接下来我们讨论它们在“宏观”层面所

起的关键作用，即将电路的主要部分连接在一起。为此，我们需要引入一些新的术语。

模块。在构建计算机的过程中，需要构建电路以实现计算机的各种抽象组件，如我们需要构建存储器、寄存器和 ALU 等。我们使用术语模块 (module) 来指代这样的电路 (通常这些电路实现了计算机的某个基本零件，用于完成某个基本功能)。出人意料的是，在典型的计算机中，模块数量通常非常少。

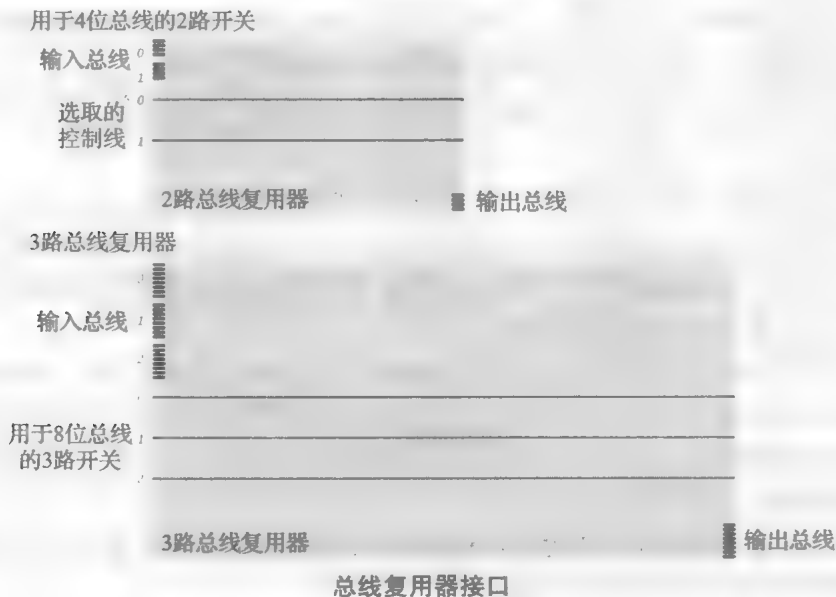
总线连接。我们使用总线 (bus) 将数据从一个模块传输到另一个模块。总线其实就是一些简单的导线组。我们有时也把总线当作数据通路 (data path)，以便明确它们的用途：在计算过程中，数据沿总线从存储器传送到寄存器，从寄存器传送到 ALU，从寄存器传送到存储器等。为了更好地使用总线，我们将输入和输出线路整合在一起，以建立总线连接 (bus connection)，而不再为每个输入输出设备各自建立独立的输入线或输出线。例如，右图展示了我们的 ALU 电路的输入和输出总线连接。像往常一样，我们把输入放在左上方，输出放在右下方。通过这些总线连接，我们可以轻松地将 ALU 连接到其他电路模块。

总线开关复用器。作为一个典型实例，我们考虑这样一个组合电路，它以  $m$  个宽度为  $n$  的总线 (总线的宽度为总线中的导线数量) 作为输入，并为整个总线实现一个逻辑开关，能够将其中一条总线切换为输出。为了选择要切换的总线，我们使用  $m$  条控制线，并假设它们中至多有一条是 1，用于选中对应于该控制线的总线并将其切换至输出。在下面图示中，上半部分画出了一个宽度为 4 的总线和一个 2 路开关的接口示意图，下半部分绘制了一个宽度为 8 的总线和一个 3 路开关的接口图。



4 位算术逻辑单元的总线连接

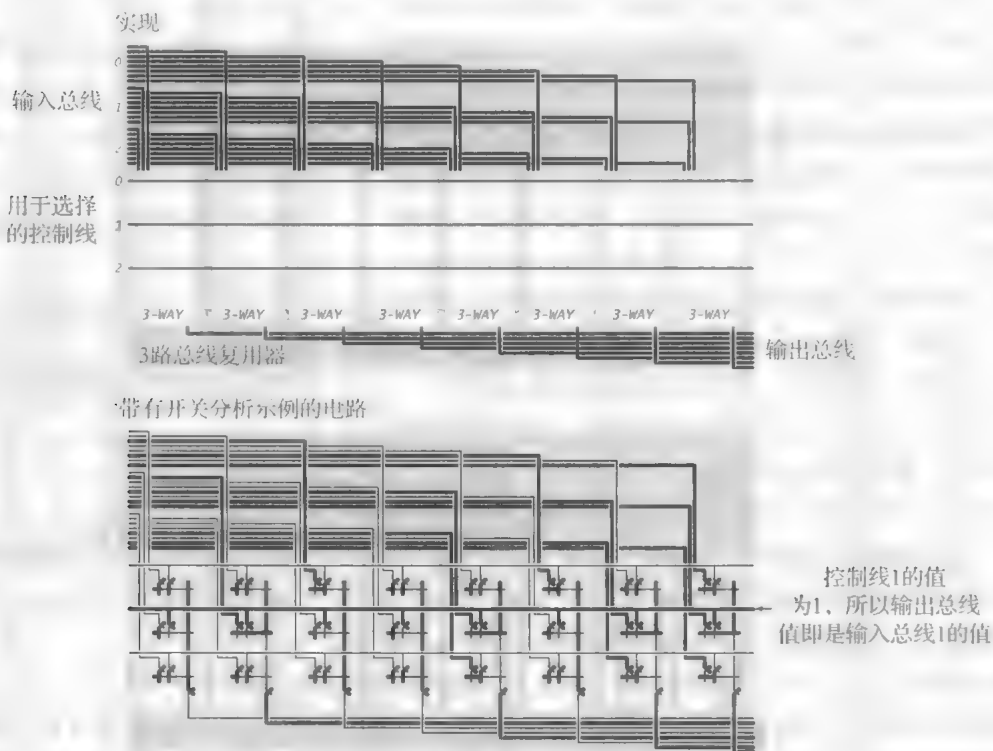
1034



总线复用器接口

1035

实现非常简单，就是对 3 路开关复用器的简单应用，总线中的每一根线都连接到一个复用器，就像我们的 ALU 底部的复用器使用一样。我们将每个位的输出线连接到一个多路选择复用器（这是我们学习的第一个门级电路），并将多路选择复用器的控制线贯穿在一起进行选择。如此一来，选中的总线对应的值将作为这个复用器的输出，出现在输出总线上。下面展示的是一个 8 位的 3 路总线多路复用器的实现，图的下方是开关分析示例。确保自己看懂了这个电路的工作原理，这一点至关重要！在 7.5 节中分析 CPU 的电路设计时，这种连接和控制的方式也会发挥重要的作用。



一个 8 位的 3 路总线复用器（实现、电路和示例）

同样，总线复用器是逻辑开关——从输入线到输出线之间没有任何物理连接。但如果不考虑电路的转换时间，电路操作就好像它们连接在一起似的。

1036

**抽象层次** 虽然我们一直是在利用电线和开关建立电路，但实际上我们已经经历了三个不同的抽象层次：

- 门 (gate) 是由开关构成的电路，如“与门”或“或非门”。
- 门级电路 (gate-level circuit) 是由门构成的电路，如 MAJ、ODD 或解码器电路。
- 模块 (module) 是由组件或门构建的电路，具有输入输出总线和控制总线以便连接到其他模块，如 ALU 或总线复用器。

事实上，我们现在已经了解了电路的内部结构，我们可以使用由总线、控制线和规定好尺寸的接口，在模块这个抽象层面构建我们的电路，如后图所示。每一个模块都将在我们的计算设备中发挥关键作用，并且我们只需要在抽象层面理解每个模块的功能和设计标准就可以完成电路的设计。

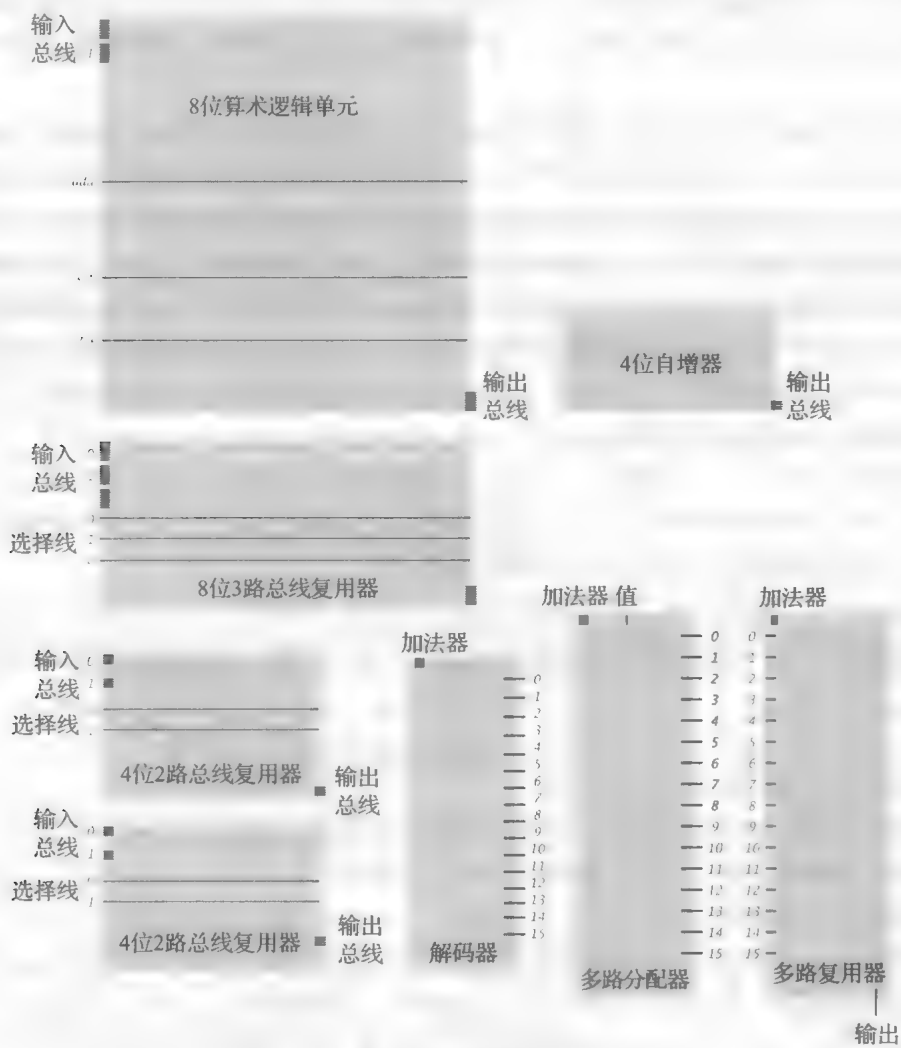
从这个层面来看，我们不必考虑如何构建模块，但是我们必须了解它们的功能。当然你应该能够理解后图中每个模块的基本功能。

我们讨论的所有的门电路都应该满足的一条属性是“输出仅取决于输入”。因此，我们所考虑的所有组件和模块都具有相同的属性：它们是组合电路。在下一节中，我们将考虑时序电路（sequential circuit），它们的物理状态（和输出）取决于输入值的变化。

就像在开发软件时一样，我们在构建电路时也会将电路进行抽象。这有助于我们将问题简化，并且它可以将功能实现部分和使用接口设计进行分离，这与构建软件有着诸多的相同点。只要模块的使用者和功能实现部分的外部接口（也就是输入线和输出线）保持一致，我们就可以独立地测试和调试模块，更好地实现功能并能够复用模块。

布局。电路比软件更接近于物理世界。其实在定义电路的尺寸和输入输出线的位置时，我们已经在接口中包含了布局信息。我们总是利用现有的物理模块建立一个更高级别的电路，而这种模型表示方法很直观地表达了这种工作方式。只要模块的输入线和输出线处于同一位置，我们就可以拔下一个旧模块并插入新模块。

1037



组合电路模块接口

1038

如果我们完全忽略布局约束，组合电路和布尔函数之间几乎没有什么差别。就像我们在加法器中所用的 MAJ 函数和 ODD 函数，它们的布尔方程其实与对应电路中所用的门电路以及元件是等价的。布尔函数的表示方法非常严谨，也是很有价值的，但是它们与物理电路

相差甚远。我们使用的带有布局信息的抽象表示方法是一个折中方案。在进行电路设计的过程中，有时我们可能完全忽略抽象层的设计，只关注于开关、导线这些实现的细节；有时我们可能会使用完全的抽象表示法，然后将模块的布局和摆放视为另外一个独立的任务。我们的方法将电路分为两个部分：一个是布尔函数部分，一个是物理电路部分（这一部分你只能在撬开计算机后才能看到），然后又把这两部分有机结合起来。

如果你还不能理解我们之前所讲的在几个不同的抽象层次构建电路的方法（包括真值表、积之和表示法、MAJ 和 ODD 函数以及它们如何对应到加法电路设计中的应用），那么再试图理解 ALU 电路如何工作就会变得十分困难。同时，理解物理电路的工作原理也很重要。对于每个特定的计算，在开关和导线级分析哪些线的值是 1，哪些线的值是 0，是非常有意义的训练，它可以让你对如何将算法构建成具体的电路有更加细致而深入的理解。

我们抽象出的电路模型是矩形，其输入和输出位于矩形的边，但对于我们来说这种抽象的思想远比这些抽象模型本身重要。我们也可以考虑开发其他不同形式的抽象模型，比如输入和输出可以出现在电路中任何地方，或者将电路画成三维的，模块表示成平行六面体，等等。

[1039]

如果我们想构建一台计算机，那么我们就一定要把之前所设计的那个抽象计算模型在物理世界中建造出来。如果我们在物理世界中可以轻易地制造出容纳指数级门电路的芯片，那么我们就可以简单地用积之和电路来构建所有的组合电路；如果很容易决定如何布置模块和电线，那么我们就可能不需要探讨电路该如何布局。我们所描述的这种抽象思想是建立在我们对物理世界的理解不断深入的基础上的。事实上，现代计算机的设计和制造都是基于电路图的，这些电路图严格规定了每根导线的位置，以及连接到导线上的每个元件的尺寸和位置，与我们所使用的图没有太大差别。这些图是生产制造处理器芯片过程的输入，而这些芯片正是现代计算机或移动设备的核心。

半个多世纪以来，当一项新技术出现时（即使是简单物理开关的新实现），人们已经能够很快地利用它来构建新的电路，并层层改进它的软件系统和应用。这种能力推动了计算应用的普及，也算是对抽象设计方法的力量最终证明。

我们在后面一节里将再次展示抽象设计方法的力量，你将看到我们使用十几页的内容来描述一个完整的 ALU 电路，既能清楚地表示它的功能和逻辑，也能够描述电路的基本属性。

[1040]

## 问答环节

问：真正的计算机也是这样设计的吗？

答：我们在设计的过程中忽视了许多物理限制。但是，过去绝大多数的计算机基本上都是按照这样的逻辑设计的，现代计算机也基本上是这样的。

问：这样做有什么缺陷吗？

答：我们这么做的目的是使你理解电路，而非设计电路。如同在学习 Java 的过程中，最开始只是让你读懂一些程序，而非自己设计程序。当你编程时，面对一张白纸你可能会写下一些 Java 代码，但未必如你想象得那样简洁和准确。这同样适用于抽象层和接口的设计。一旦我们固定一个接口，我们就会发现它在设计中的问题；我们可能需要仔细地进行大量的重复验证，比较不同设计方案的优劣，最终得出一个有效的接口设计规范。与软件一样，我们需要制定许多设计规则，在电路设计中你必须遵守它们。

问：香农的名字似乎之前出现过？他是做什么的？

答：香农是信息论之父。信息论是研究信息表达的学科，它是通信机制、数据表达，以及电子信息时代其他许多重要领域的基础。

问：我的计算机里的加法器是一个波纹进位加法器吗？

答：可能不是。现代计算机通常会使用一个更复杂的模型，以加快处理进程，其处理时间与位数的对数成正比。

问：为什么不使用多路复用器来实现总线多路复用器呢？

答：其实是可以的。但是通常情况下我们的控制线是独立的，而不是二进制编码的，所以，本节中讲解的这个模型更方便一些。参见练习 7.3.16。

1041

## 练习

7.3.1 在一个输入 / 关闭开关里，假设输入值为  $x$ ，控制值为  $y$ ，写出输出的布尔表达式。

答案：使用“与非”表达式： $xy'$ 。根据我们的规则，以下两个电路是相等的（当且仅当  $x$  为 1， $y$  为 0 时，输出值为 1）：



我们使用左边的图，以确保输入的对称性，并能避免混淆哪个导线是控制线。在实际的电路设计中，我们会尽可能避免将值在输入和输出间直接传递，以保证所有电路输出都取决于相邻的电源点，而不是从输入端产生的电流。你可以将我们前面使用到的扩展的或非门看作一连串这样的电路连接到一个非门上。

7.3.2 根据上一题的思路，设计一个只有两个开关的与门。

答案：将上面电路的  $y$  值取非。根据我们的规则，当且仅当  $x$  和  $y$  均为 1 时，电路输出值为 1。



7.3.3 仅使用与非门（详见练习 7.3.1 的右图），如何构建非门、或非门、或门、与门和扩展的与门。

7.3.4 设计一个 3 路的开关电路，当且仅当一个选择线的值为 1 且对应的输入为 1 时，输出为 1。与 7.3 节正文电路不同：当有两个输入线的值为 1 且两个选中线为 1 时，输出应该为 0。

7.3.5 描述以下操作的结果：将一个多路分配器的输出与一个同规格的多路复用器的输入连接，并将它们的地址线连接到相同的输入线上。

1042

7.3.6 证明： $\text{MAJ}(x, y, z) = xy + xz + yz$ ，并设计一个只有 3 个双输入的与门、2 个双输入的或门的电路以计算 MAJ。

7.3.7 设计一个三输入 ODD 的电路，使用少于 5 个门。

7.3.8 设计两个电路，用双输入的 NAND 门实现双输入的 XOR 函数。在第一个电路中，可以将任一 NAND 门输入连接到 1 或者 0。在第二个电路中，所有 NAND 门的输入都需要连接到电路输入或者另一个 NAND 门的输出。

7.3.9 画出真值表以证明：

$$\text{MAJ}(x, y, z) = (x, y, z)(x' + y + z)(x + y' + z')(x + y + z')$$

7.3.10 证明：每一个布尔函数都可以表示为一个和之积电路，其中求和部分可以是输入信号或者输入信号取反，就像上一题的表达式那样。



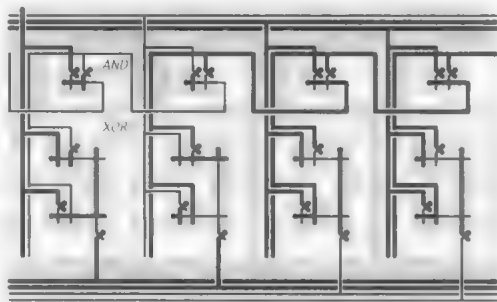
- 7.3.11 基于上一练习，设计一个和之积电路方法，并用这个方法为 ODD 设计一个电路。
- 7.3.12 为练习 7.1.12 中三参数的多路复用器设计一个积之和电路，并为相同函数给出一个仅使用 3 个门的电路实现。
- 7.3.13 使用上一练习中的 3 门电路，在输出的基础上再设计一个电路，该电路将它的  $n$  位输入顺序反转后输出。注意：还可以使用同样的思路设计出一个计算任意输入序列的反转电路。
- 7.3.14 设计具有四个输入参数的门电路，将前两个参数 ( $w, x$ ) 和后两个参数 ( $y, z$ ) 分别作为一个 2 位的二进制数字，当且仅当  $wx > yz$  时，输出为 1。
- 7.3.15 修改正文中的加法器电路，将原先的进位输出替换为溢出输出：如果最左边的进位输入与其进位输出相同，则输出为 0；若不同，则为 1。并证明，这样的修改同样适用于  $n$  位二进制补码数字表示法。
- 1043 7.3.16 设计一个  $2^n$  路总线多路复用器，其选中的输入为  $m$  条线，输入的值为选中线的二进制编码。
- 7.3.17 假设一个开关完成其开关状态切换所花费的时间为  $t$ （整个导线上立刻就能体现出值的变化）。计算以下电路在输入值变化时完成全部的状态反应所需的最长时限。

- |          |             |
|----------|-------------|
| a. NOR   | b. NAND     |
| c. 3-WAY | d. DECODE   |
| e. MUX   | f. MAJ      |
| g. ODD   | h. $n$ 位加法器 |

- 7.3.18 设计一个 4 位宽的自增电路。

答案：修改正文中的加法器，使其进位输入为 1：设  $x_3x_2x_1x_0$  为数字， $c_4c_3c_2c_1$  为进位， $z_3z_2z_1z_0$  为输出。当且仅当  $x$  位值为 1 且有进位输入为 1 时，则向下一位进位（即与操作）；当且仅当  $x$  位值和进位输入均为 1，或者均为 0 时，输出为 0；如果一个为 0，一个为 1，则输出为 1。因此，我们可以使用加法器的电路略做修改，用与门作为进位逻辑，用 XOR 门构建求和部分。

$$\begin{aligned}
 c_1 &= \text{AND}(x_0, 1) \\
 c_2 &= \text{AND}(x_1, c_1) \\
 c_3 &= \text{AND}(x_2, c_2) \\
 c_4 &= \text{AND}(x_3, c_3) \\
 z_0 &= \text{XOR}(x_0, 1) \\
 z_2 &= \text{XOR}(x_1, c_1) \\
 z_1 &= \text{XOR}(x_2, c_2) \\
 z_3 &= \text{XOR}(x_3, c_3)
 \end{aligned}$$



一个 4 位波纹进位增量器

1044

## 创新练习

- 7.3.19 设计一个门。开发一个 Java 程序，使其可以设计一个  $n$  位扩展的与门电路，与正文中描述的设计相同。从命令行中输入一个小于  $2^n$  的非负整数，用这个整数的二进制表示方法中 1 的位置来指定取反的位置。
- 7.3.20 设计一个解码器。开发一个 Java 程序，使其可以设计一个  $n$  位的解码器。可以在上一题答案的基础上开发你的程序。
- 7.3.21 设计一个积之和电路。开发一个 Java 程序，使其可以设计一个  $n$  位的积之和电路。从命令行中输入一个在 0 和  $2^{2^n}$  之间的一个整数，使用这个整数的二进制表示方法来指定给出函数值的真值表列。

7.3.22 设计一个加法器。开发一个 Java 程序，使其可以设计一个  $n$  位的加法器，与正文中描述的设计相同。可以在上一题答案的基础上开发你的程序。

7.3.23 比较器。设计一个电路，有两个  $n$  位的输入和 1 位输出，将两个输入看作二进制整数，当第一个小于第二个时输出为 1。画出  $n = 4$  时的电路，并给出一个  $n$  的函数用于计算实现该电路所需的门的数量。

7.3.24 门的通用集。门的通用集是指：若仅使用导线和集合中的门就可以实现所有布尔函数的电路表示，那么这个集合就是通用的。我们的积之和电路构建方法表明，广义的多路门都是通用的（这没什么惊喜的，因为有这么多种类型的门）。假设：非门仅有一个输入，而所有其他的门都只有两个输入，那么以下选项中哪个是通用的？并证明其余选项不是通用的。

a. NOT and AND

b. NOR

c. NAND

d. AND and OR

e. NOT and OR

f. AND and XOR

1045

7.3.25 开关的通用性。证明：在组合电路中开关和门是等价的，即对于任意给定的由电线、通断开关、输入 / 关闭开关组成的网络，可以建立一个由门构成的计算网络，其在相同的输入时总是产生相同的输出，反之亦然。提示：将上一题解法中加入与非门。

7.3.26 右移位。构建一个 4 位的右移位电路。假设输入线为  $x_0x_1x_2x_3$ ，输出线为  $z_0z_1z_2z_3$ ，控制线为  $s_0s_1s_2s_3$ ，当控制线中仅有一个为 1 时，该线所在的位置对应的数值作为移位的参数将输入数据右移。基于以下四个布尔公式来完成你的设计。

$$z_0 = x_0s_0$$

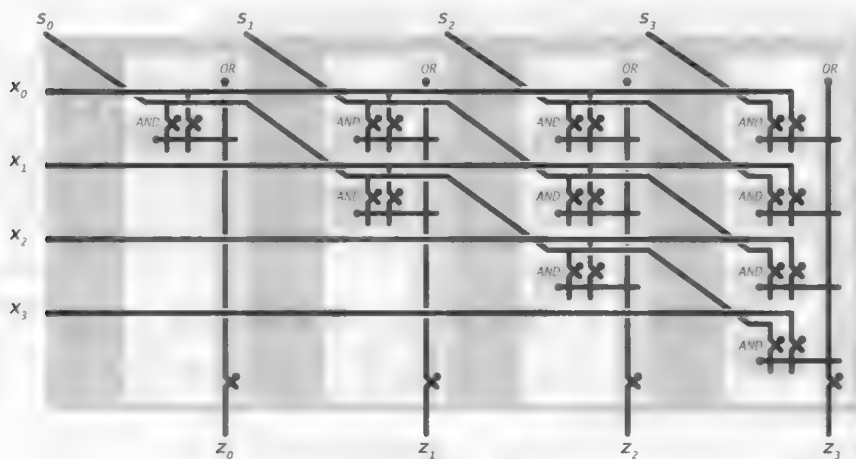
$$z_1 = x_0s_1 + x_1s_0$$

$$z_2 = x_0s_2 + x_1s_1 + x_2s_0$$

$$z_3 = x_0s_3 + x_1s_2 + x_2s_1 + x_3s_0$$

其中，你可以轻易证明：当  $s_0$  作为选中的控制线时， $z_0z_1z_2z_3$  即为  $x_0x_1x_2x_3$ ；当  $s_1$  作为选中的控制线时， $z_0z_1z_2z_3$  为  $0x_0x_1x_2$ ；当  $s_2$  作为选中的控制线时， $z_0z_1z_2z_3$  为  $000x_0$ 。

答案：



一个 4 位右移位

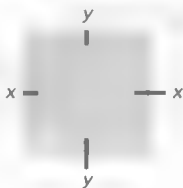
1046

7.3.27 右移位电路分析。将练习 7.3.26 中的右移位电路做一个备份，并分析这个电路计算  $1001 \gg 2 = 0010$  的过程，将计算时值为 1 的导线加粗标出。

7.3.28 左移位。根据上一题中描述的右移位的方法，给出一个左移位的布尔公式，其中：输入为

$x_0x_1x_2x_3$ , 左移位选择  $t_0t_1t_2t_3$ , 输出为  $z_0z_1z_2z_3$ 。注意到上题的电路图中对角线下方几乎是空白的, 如何在这个电路图的空白区域添加部分电路, 就能构建一个可以向任一方向移位的电路。假设 8 条控制线  $s_0s_1s_2s_3$  和  $t_0t_1t_2t_3$  中至多有一个为 1。

- 7.3.29 算术移位。对于正整数, 右移位与被 2 的幂次方相除是一样的; 左移位与被 2 的幂次方相乘是一样的。设计一个移位器, 可以对任意二进制整数 (可以是正数也可以是负数) 计算乘法或者除法, 假设数据使用二进制补码法表示, 电路应包括一个输出, 用于表示溢出。画出当  $n = 4$  时的电路图, 在设备级抽象表示即可。
- 7.3.30 位的统计。设计一个电路, 有  $n$  位输入, 并将输入中 1 的个数的二进制表示方法作为输出。画出  $n = 4$  时的电路 (门级抽象), 并给出一个  $n$  的函数, 用于计算实现该电路所需的门的数量。
- 7.3.31 乘法器。交替地使用移位和加法, 就能够实现两个  $n$  位二进制数字的乘法。按照这个原理可以设计乘法器电路。画出  $n = 4$  时的电路 (组件级抽象), 并给出一个  $n$  的函数, 用于计算实现该电路所需的门的数量。
- 7.3.32 平面计算机。在前面的电路绘制过程中, 你需要在一个平面上将多个门用导线连接起来, 而导线之间是不可以交叉的。如果不得不产生导线交叉, 可以使用一个专用的电路模块来实现, 这个模块的内部没有导线交叉, 完全使用门电路来实现, 它的功能是输入信号  $x$  和  $y$  分别在模块的左侧和上方, 而右侧的输出值为  $x$ , 下方的输出值为  $y$ 。画出这个模块的实现。



1047

## 7.4 时序电路

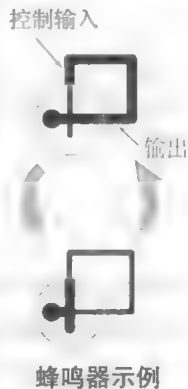
在本节中我们将会看到当电路有回路或反馈 (feedback) 时将会发生什么。实际上, 我们只会考虑一些简单的例子, 因为由反馈引入的复杂性比我们在 Java 程序中添加循环和条件引入的复杂性还要大。因此在电路中, 我们严格限制反馈电路。

首先, 我们只考虑最简单的门电路上形成的反馈, 由两个开关形成最基本的回路电路。值得注意的是, 这样的回路使得我们可以构建一种全新的电路类型——内存 (memory)。我们会重点关注如何为这些电路添加控制线, 从而精确控制存储的数据值。在本节中我们考虑的所有电路都仅限制在单个基本循环中进行反馈。理解反馈的机制是理解计算机内存实现原理的关键。存储器以字节形式存储, 字节由位构成, 而位是由反馈回路实现的。

在下一节中, 我们讲解大型电路元件之间的大型反馈回路 (macro feedback loop)。最终, 我们的计算设备中只可以有少数这样的循环, 并且我们能够准确地控制它们的行为。

**基本反馈电路** 我们首先来分析两个可能是最简单的反馈回路。第一个只有一个开关; 第二个有两个。

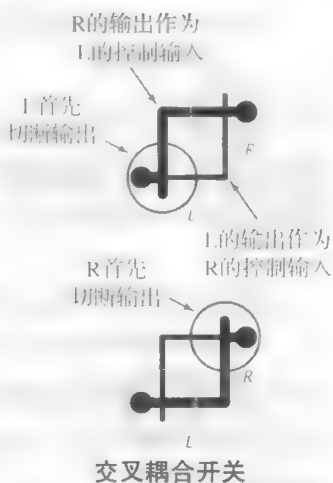
**蜂鸣器。**我们来分析右侧所示的电路, 其中开关的输出被反馈到其控制输入端。如果输出是 1 (顶部), 那么控制输入也是 1, 这会导致开关将输出变为 0。但是一旦开关完成这个工作 (底部), 控制输入也变成 0, 这导致



开关再次将输出变为 1 (顶部)。这是一个不稳定的情况, 因为开关始终处于一个循环当中, 不停地翻转输出的值 (以及控制输入的值), 而且翻转速度就是物理设备的最快速度。这样的电路被称为蜂鸣器 (buzzer)。旧式门铃就是由这样一个模块配合继电器开关构成的: 继电器的输出端连接到控制输入端, 电磁铁触点来回摆动会产生嗡嗡声。

1048

**具有反馈的稳定电路。**例如右边的反馈回路, 它由两个互联的开关组成, 每个开关的输出端都连接到另一个开关的控制线。我们将这种连接方式定义为交叉耦合 (cross-coupled)。如图所示, 交叉耦合回路具有两种形式, 具体取决于开关的操作顺序。假设电源关闭 (所有电线的值均为 0), 然后打开开关。上面的图描述了当开关 L 在开关 R 之前接通 (即使只有很短的时间), 将会发生什么情况: 按照其控制线的控制, 开关 L 将其输出设置为 0, 这意味着开关 R 的控制是 0, 因此开关 R 的输出是 1, 这使得开关 L 保持工作状态 (将其输出设置为 0), 这是一个稳定的状态。下面的图描述了当开关 R 中的电源首先被打开的情况, 你可以按照相同的逻辑分析将会发生什么。无论哪种方式, 电路都是稳定的: 只要第一个开关连接到电源, 什么都不会改变。我们期望所有的电源都在同一时刻开启, 但实际上我们无法控制这样一个电路可能到达的状态。我们要做的第一件事就是增加控制线来纠正这种情况。



这两个例子说明了当我们在开关电路中引入反馈时产生的两种具有本质区别的模型。构建复杂电路需要更多的精力, 所以我们要确保能够理解和掌握这些较小的组件, 并基于它们构建更大的电路组件, 如同构建组合电路的过程一样。幸运的是, 正如你将在本节中看到的, 通过在最简单的稳定状态电路 (即两个交叉耦合开关) 上添加控制线, 就可以提供足够的灵活性, 使我们能够为计算机构建存储器电路。

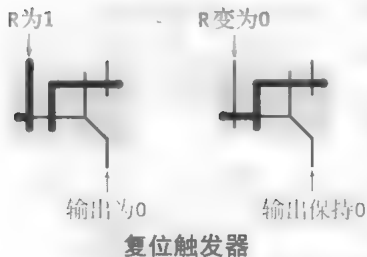
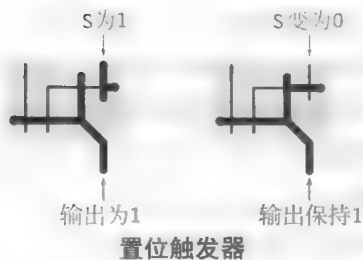
**触发器** 通过在我们的交叉耦合电路中增加两个控制输入, 我们可以创建一个触发器 (flip-flop), 这就是我们的一个内存位的基本实现。控制线的用途正如它的名称一样: 控制电路到达两种状态中的哪个。当我们希望电路选择其中一个状态时, 我们激活相应的控制线; 当我们想要在另一个状态时, 我们激活另一个控制线。当然, 我们可以这样理解: 其中一个状态理解为 0, 另一个状态就表示 1, 这样通过这种方式添加两条控制线就可以实现一个内存位 (a bit of memory), 我们可以借助控制线将它设置为 0 或 1。

上述描述的功能其实很好实现, 如右图所示。我们在每个电源节点的前面添加一条控制线, 并在底部输出一个输出值。按照惯例, 我们将控制线分别标记为 S (用于置位, 来自于 Set 的首字母) 和 R (用于复位, 来自于 Reset 的首字母)。通常, 我们使用交叉耦合的 NOR 门来实现 (参见练习 7.3.1), 该电路被称为 SR 触发器 (SR flip-flop)。接下来, 我们在开关层面来分析它们的功能。

**置位。**要置位一个触发器 (输出值为 1 的稳定状态), 我们只需将 S 控制线的值设为 1 的时间足够长, 足以打开阻塞与右侧电源节点相连接的开关, 这会使输出线的值设为 1, 这个值为一直保持下来, 即使控制线 S



1049



值变为 0，如左图所示。一定要理解这点，这是理解计算机如何存储信息的关键。

**复位。**要复位一个触发器（输出值为 0 的稳定状态），我们将 R 控制线的值提高到 1 的时间足够长，足以打开阻塞与左侧电源节点相连接的开关，这会使输出线的值设为 0，这个值为一直保持下来，即使控制线 R 值变为 0。这种情况如左图所示。

我们允许 R 和 S 控制信号都是 0，但是我们并不希望它们都置为 1，这将导致我们无法控制输出的值。这种情况被称为竞争条件（race condition），因为两个开关将比赛谁会第一个被输出。我们在电路设计中应该避免这样的情况出现。

触发器的意义不仅仅在于它们很简单，更在于每一个触发器都实现了内存位，这是计算中最基本的抽象单元。实际

上，这种内存位的实现非常简单，令人难以置信的是，几十年来，触发器一直是计算机内存的基础模块。接下来，我们加入能控制内存位输入的控制线，来实现计算机处理器的寄存器和主存储器部分。

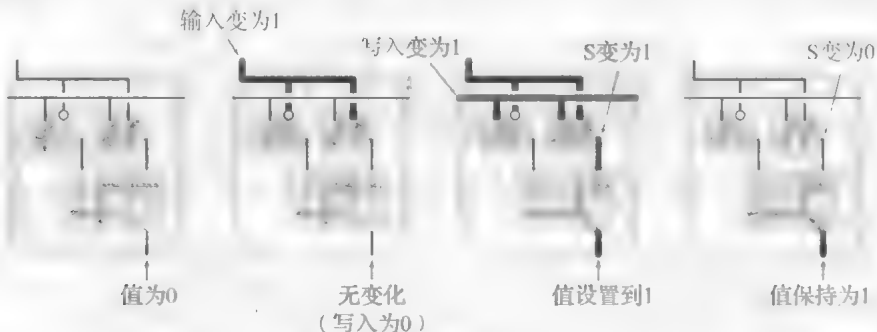
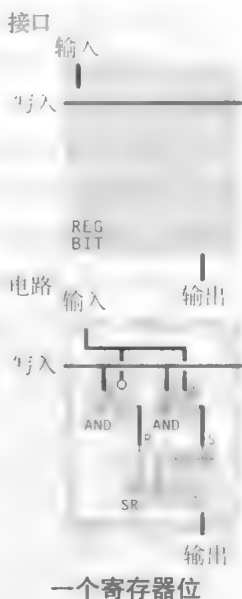
1050

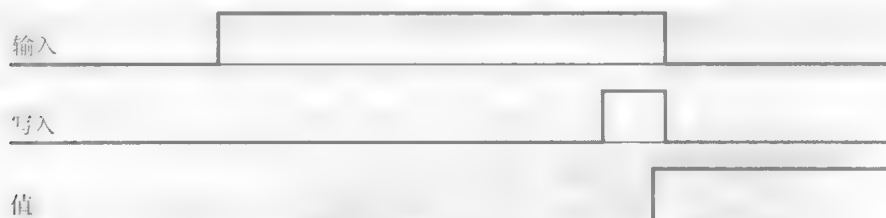
**寄存器** 在下一级抽象中，我们增加了两条控制线来构建基于触发器的寄存器位（register bit）。然后，我们将一系列寄存器位串起来作为一个寄存器（register）。寄存器是每台计算机的重要组成部分；就像 TOY 机中的各个寄存器，如 PC 或 R[0] 至 R[F]。

**寄存器位。**我们的目标是实现以下功能：首先，我们希望通过一条独立的输入线来传输想要存储的值；其次，我们希望有一个写入（write）控制线，以精准控制在某一个时刻当它的值变化时，我们将输入的值保存下来。右图给出了实现电路，这个电路控制 S 和 R 信号，如下所示：

- 当且仅当写入控制线是 1 并且输入线是 1 时，S 是 1。
- 当且仅当写入控制线为 1 并且输入线为 0 时，R 为 1。

我们会按照统一的方式设置寄存器中的值。其中时序的控制非常重要，如下图所示。首先，我们确保输入线上存在输入值。然后将控制线置 1，但是置 1 时间不用很长，只要能够保证开关操作可以置位或复位触发器即可。



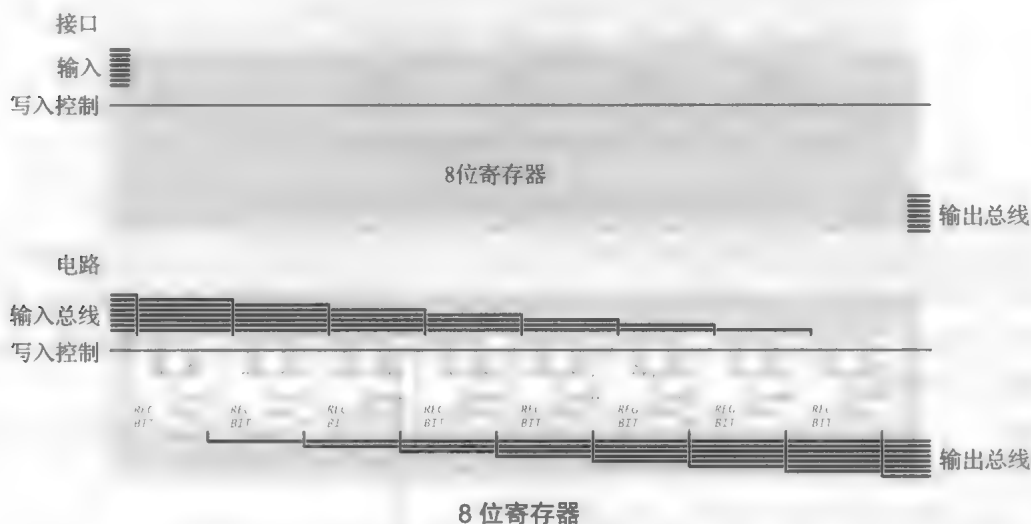


将寄存器置位为 1 (续)

1051

我们需要控制时序的原因是，我们必须考虑到触发器值的变化可能影响输入线上值变化的这种可能性，比如在一个包含大量线路和开关的长反馈回路中就可能出现这种情况。事实上，这种反馈是我们计算设备的一个重要组成部分。幸运的是，这种写入控制机制足以让我们在电路中解决这个问题。

寄存器。寄存器位使我们能够实现寄存器 (register)。下面是一个 8 位寄存器的例子。它包括左上方的 8 条输入线、右下方的 8 条输出线和一条贯穿寄存器的写入控制线。对于接口部分的实现，我们只需并排放置  $n$  个寄存器位，将它们的输入端连接到顶部的总线上，将它们的输出端连接到底部的总线上，并将写入连接到一起，使控制线能够贯穿每一位。



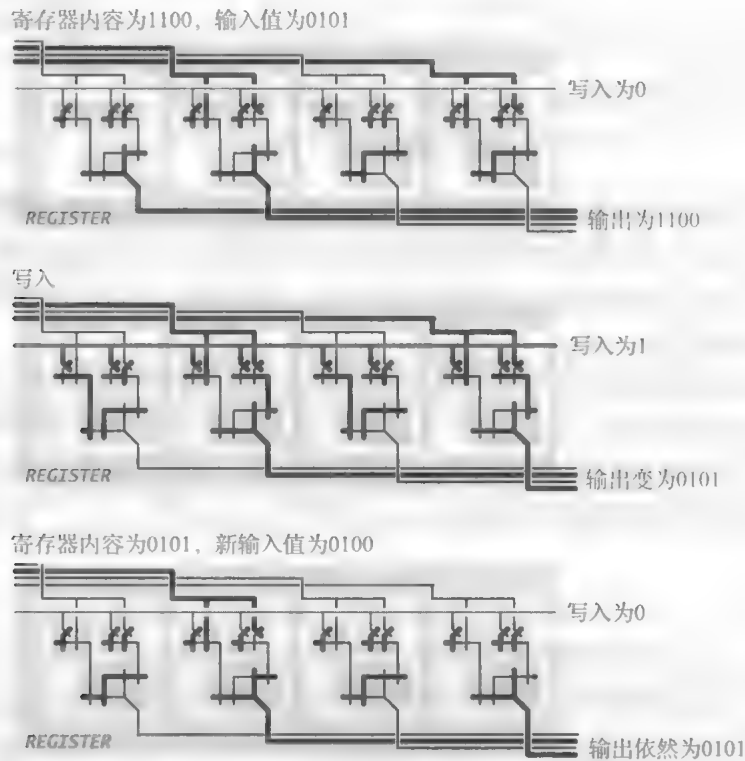
8 位寄存器

寄存器的写入。按照前面描述的内存位的写入脉冲 (write pulse)，我们以完全相同的方式控制寄存器的时序，如下图中所示的 4 位寄存器。在上方图中，寄存器保存的值是 1100，同时，输入线上的值是 0101。由于写入控制线是 0，输出线仍保持值 1100，即寄存器存储的内容。当写入脉冲 (很短暂) 变为 1 时，在中间的图中，新值 0101 被存储在寄存器中并立即出现在输出线上。在最下方的图中，当写入脉冲变为 0 时，即使输入线上有不同的值，寄存器和输出线上的值仍保持为 0101。新的值将不会被加载，直到写入控制再次变为 1。

1052

在大多数计算机的处理器中，寄存器都起着重要的作用。它们存储着内部处理器的信息，就像 TOY 计算机中的程序寄存器 PC 或指令寄存器 IR；它们还用于实现程序算术运算的寄存器，如 TOY 中的 R[0] 至 R[F]。在我们的电路中，两种情况的实现和操作是相同的。每个寄存器在其触发器中保存值，且这些值都可以在输出线上可用。当写入脉冲到来时，输

入线上的值被存储在寄存器触发器中。你将会看到，这个简单的接口足以让我们实现一个典型的计算机处理器的所有功能。



一个 4 位寄存器，在写入脉冲时的开关状态分析

1053

**内存** 我们的下一个时序电路是内存（memory）。从硬件实现的角度来看，内存只是一个可寻址的寄存器序列。如果我们的字大小是  $n$ ，并且在我们的计算机指令中使用  $m$  个地址位，那么我们有  $2^m$  个内存字，每个字包含  $n$  位。内存字的原理类似于一个寄存器，只是附加了寻址 / 选择机制。

**接口。**一个内存有  $m + n$  个输入和  $n$  个输出，输入包括  $m$  个地址位和  $n$  个输入位。写入控制信号线控制写入的时序，就像寄存器一样。我们希望它有如下功能：

- 地址字的内容始终在输出线上有效。
- 在写入控制线上产生有效信号脉冲时，输入线上的值被存储到地址字中。

可以看到，一个这样的内存中会拥有  $2^m$  个字，存储量相当大，因此存储相关的模块占据了芯片很大的一部分。

**位片设计。**我们的设计是经典的位片内存（即内存中所有字的同一个位会放在一起处理——译者注），我们在每一个位的位置上使用相同的电路：用多路分配器选择一个位进行写操作，用多路复用器选择一个位进行读操作，如后图所示（你可能需要回顾前文中的多路分配器和多路复用器相关内容）。每个位的位置垂直方向依次排列，左边是多路分配器，右边是多路复用器。相同的输入地址同时驱动多路分配器和多路复用器，因此只有在地址字中的位才是有效的（换言之，它会受输入的影响或影响输出）。将  $n$  个这样的电路组合在一起形成一个  $n$  位字的存储，应注意此时只有被寻址的字才是有效的。

内存仅有一个写入控制线，该线用作多路分配器的输入，并被切换到被选中的地址字上

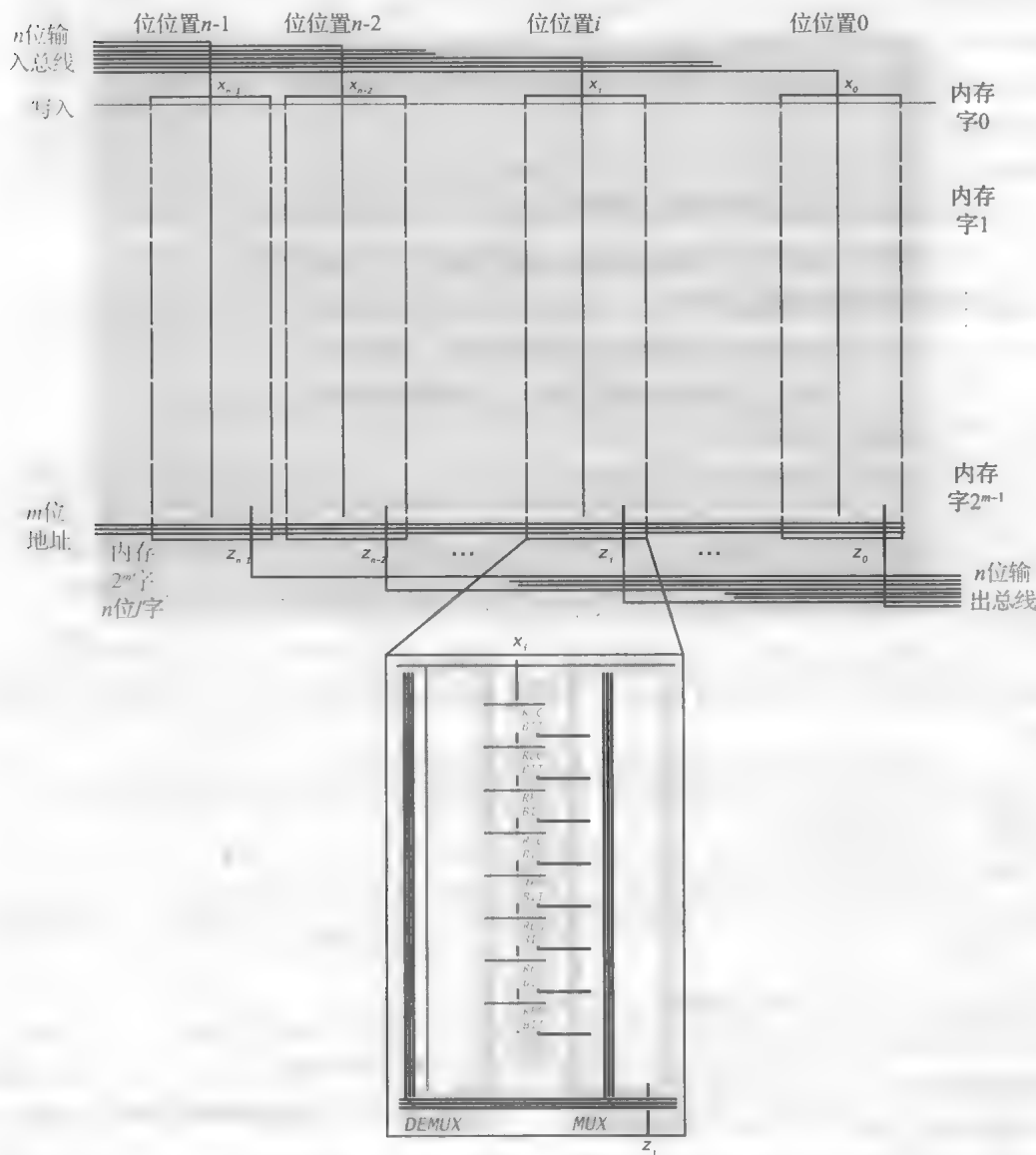


每位的写入控制线。当一个写入脉冲传入内存中时，即会传输到地址字的每位（而且仅有那些位），从而使得输入值可以对存储单元进行置位或复位。内存值随时可以被读取，因为每位的多路复用器总是可以输出被寻址位的触发器的输出值。

电路实现。由于存储单元占用太多的空间，我们用简化的设计来构建电路：

- 我们不再为每位都配置一个多路分配器，而是总共配置一个，并使得每一个输出都水平地穿过内存字的每一位。
- 我们将多路复用器的与门集成到寄存器位中，从而使得每一个内存位都具有了自己的选通控制线（select control line）。

1054



位片内存示意图

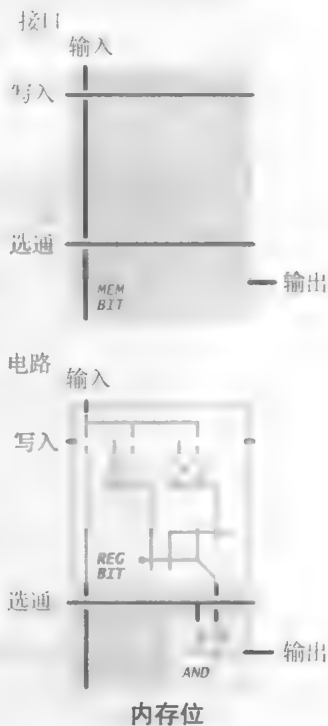
1055

这些简化的设计使得一个“位片”的宽度缩减为之前设计所需宽度的约五分之一（这是非常有必要的，因为你将会看到，即使这样它仍然占据了处理器设计图的一整页纸）。这样的设计实现了与之前完全相同的内存接口，因此了解这些通信的细节变得不那么重要。如果你有兴趣，可以在本节最后的问答环节中找到一些答案。

内存位。如前所述，我们的内存是由内存位（memory bit）构成的，这些内存位是寄存器位加上选通控制线（用于读取）构成的。如果选通控制线本身的值为 0，则输出为 0；如果选通控制线为 1，则输出是触发器的值。为了实现这种行为，我们只需将触发器的输出与选择线进行“与”运算来计算内存位从而进行输出，如右图所示。对于每位的位置，这些输出被传送到垂直或门，多路复用器的功能就是接收这些位的值并将其中一个作为输出值。选通线的作用是确保除了被选取的那条线外，其余线的值均为 0。

内存电路。我们用构建寄存器相同的方式构建每个内存字：将每位水平摆放，并把它们的控制线连接起来，使得控制线贯穿内存字的每位。然后通过垂直堆叠内存字来建立内存。后文图展示了一个具有四个 6 位字（ $m = 2$  和  $n = 6$ ）的内存的实现。每个字的地址为两位即 00、01、10 和 11。应该注意到这个电路有以下特点：

- 每位对应于一条输入线，从顶部的总线引出来垂直贯穿这一个位片。
- 一个电路同时实现了解码器和多路分配器功能（只有被寻址的那两个输出是有效的）。
- 写入控制线和地址线驱动解码器 / 多路分配器，其输出针对每个内存字的写入线和选通控制线，这两条线会水平贯穿每个内存字的每位。
- 一个垂直或门用来采集每位的输出。
- 底部的总线用于输出，每位对应一行。



1056

为了更好地理解内存的写入操作，字 10 在该图中被突出显示（并且展示了每位的电路细节）。在这个例子中，输入总线的值为 001111，地址位为 10，写入控制线为 1，因此内存字 10 的写入和选通线都是 1，它将触发器的值设为 001111（从右向左读），并将这些值传递到垂直 OR 门中并最终传送到输出总线。

总之，这种内存设计产生的电路具有  $m$  个地址输入、 $n$  位输入和输出总线，以及一个写入控制线。它的功能如下：

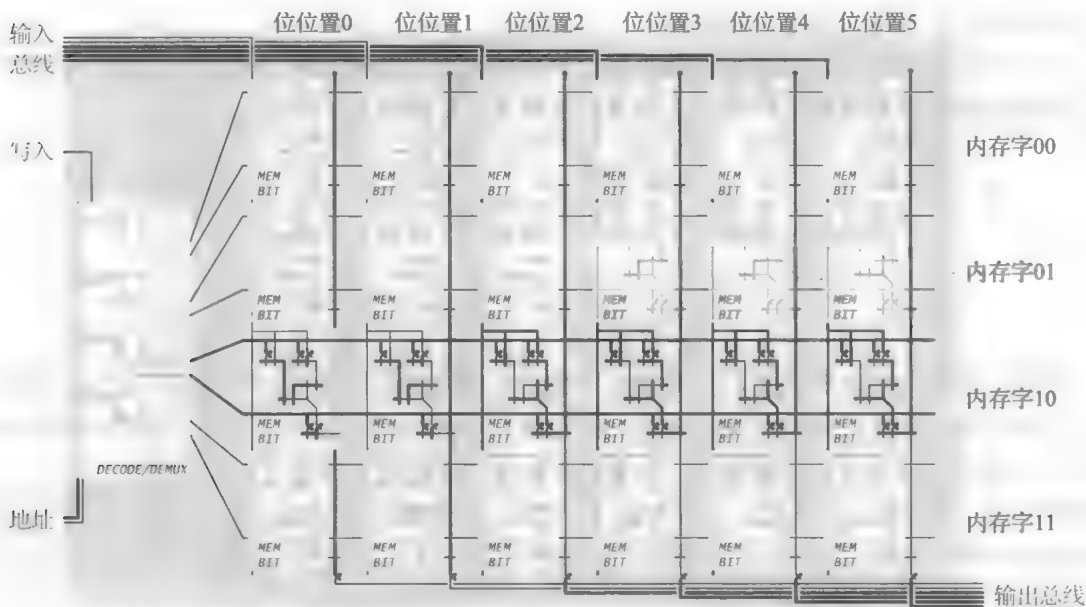
- 地址字的内容总是会出现在输出总线上。
- 每次产生写入脉冲，输入总线值都存储在地址字中。

右边的表格展示了实现这样的内存所需的门的数量与这些参数的关系。需要设计计算机电路的每个人都能很快意识到，每个内存位能否被高效而紧凑地实现是整个电路设计中非常重要的。

| 字        | 位/字 | 触发器      | 基他门的总数 |
|----------|-----|----------|--------|
| 4        | 6   | 24       | 82     |
| 8        | 8   | 64       | 208    |
| 256      | 16  | 4 096    | 12 562 |
| $2^{30}$ | 64  | $2^{36}$ | 大约700亿 |

不同规格的内存中门的数量

时钟 我们最后一个时序电路的例子是时钟（clock），它是计算机中一个重要组成部分。“时钟节拍”（ticking clock）是一个基本的抽象，它将计算电路中发生的所有事情同步化。我们的电路是这个抽象的硬件实现。时钟的主要目的是驱动取指 - 执行周期，这个周期的细节我们在第一次介绍 TOY 时已描述过。事实上，我们的计算电路中发生的一切都是通过响应时钟而得到同步的。



一个内存（4个六位宽的字），写入脉冲的开关级分析

时钟滴答（tick-tock）。我们的时钟的计时方法是基于下图这样的周期性脉冲实现的。



这种脉冲信号被称为时钟信号（clock signal）。它通常是 0，但是它按照固定的周期迅速地切换到 1。我们通常把信号是 1 的时间当作一个脉冲（pulse），或者，为了表述得更加直观，我们称作一个滴答（tick）。在一台真实的计算机中，时钟信号的精度是最重要的，计算机设计师竭尽全力创造可靠、快速的时钟。如今，计算机的时钟可能会每秒触发数十亿次。

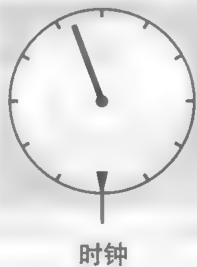
在我们的计算机设计中，假设有一个外部的信号源可以产生一个如上图所示的周期性信号，除了这一基本假设外，为了处理现实世界中的问题，我们再做两个额外的假设：

- 脉冲长到可以激活任何开关。
- 从一个脉冲开始到下一个脉冲开始的时间长于电路内最长的开关激活链。

第二个参数被称为时钟速度（clock speed），这是一个相当关键的参数，如它决定了每秒执行的指令的数量，精确的数值不是很容易计算（见练习 7.4.14）。事实上，设计现实世界中的计算机的一个常用策略是利用经验确定速度：从一个保守的长时间间隔开始，不断地给时钟加速，直到时间不足以激活所有开关而导致电路中断，然后将时钟时间设置得相对慢一点。

通常情况下，计算机时钟采用不同于构建开关的技术来构建，以尽可能实现更快的时钟速度。通常，该技术涉及在物理材料内激发电气振荡。与开关一样，与物理世界的精确连接超出了本书的范围，与我们正在考虑的计算机设计的逻辑方面无关。

为了解决这个问题，你可以想象一下，在一个表盘上的接触点可以使得秒针每隔一分钟产生一个一秒钟的信号，如右图所示。这样的设备可以产生我们想要的时钟信号。在现实世界中计算机的时钟可能要快数十亿倍，但在概念上是一样的。如果选择一个合适的缩放比例，两者生



1057  
1058

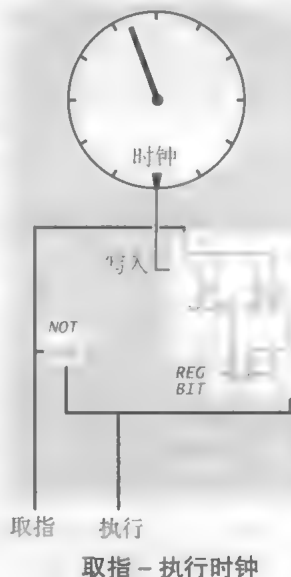
成的时钟信号看起来可能也是相同的。

取指和执行输出。时钟最重要的目的是产生周期性的控制信号，我们可以用它来区分获取指令阶段和执行指令阶段。特别是，我们想要有这样的信号：



这两个信号是相反的，所以如果我们产生一个信号来获取指令，我们可以把它连接到一个非门来得到一个执行指令，反之亦然。有很多方法可以获得这样的信号，也许最简单的就是将一个时钟连接到一个寄存器位，如左图所示。时钟控制线连接到内存位的写入控制线，内存位的输入值是存储值取反。通过这些连接，每个时钟脉冲都会使触发器对存储值进行取反运算，直到下一个时钟脉冲。我们使用内存位的值作为我们执行指令的信号以及将它的取反结果作为我们取指令的信号。触发器记住时钟的状态，直到下一个脉冲，其值从 0 变为 1 或从 1 变为 0。

写控制输出。在计算周期的获取指令和执行指令阶段，我们需要将值写入寄存器。我们会简短地分析这些具体细节，但是这个基本的实现需要我们在时钟上增加两个门电路。具体而言，我们希望再增加两个控制输出：取指写操作脉冲和执行写操作脉冲，我们希望这些信号为 0，直到它们各自对应的阶段结束时输出为 1。为了解决这个问题我们可以通过为每个阶段添加一个与门来实现。时钟脉冲与取指信号进行“与”操作得到取指写操作脉冲，同理，时钟脉冲与执行信号进行“与”操作得到执行写操作脉冲。



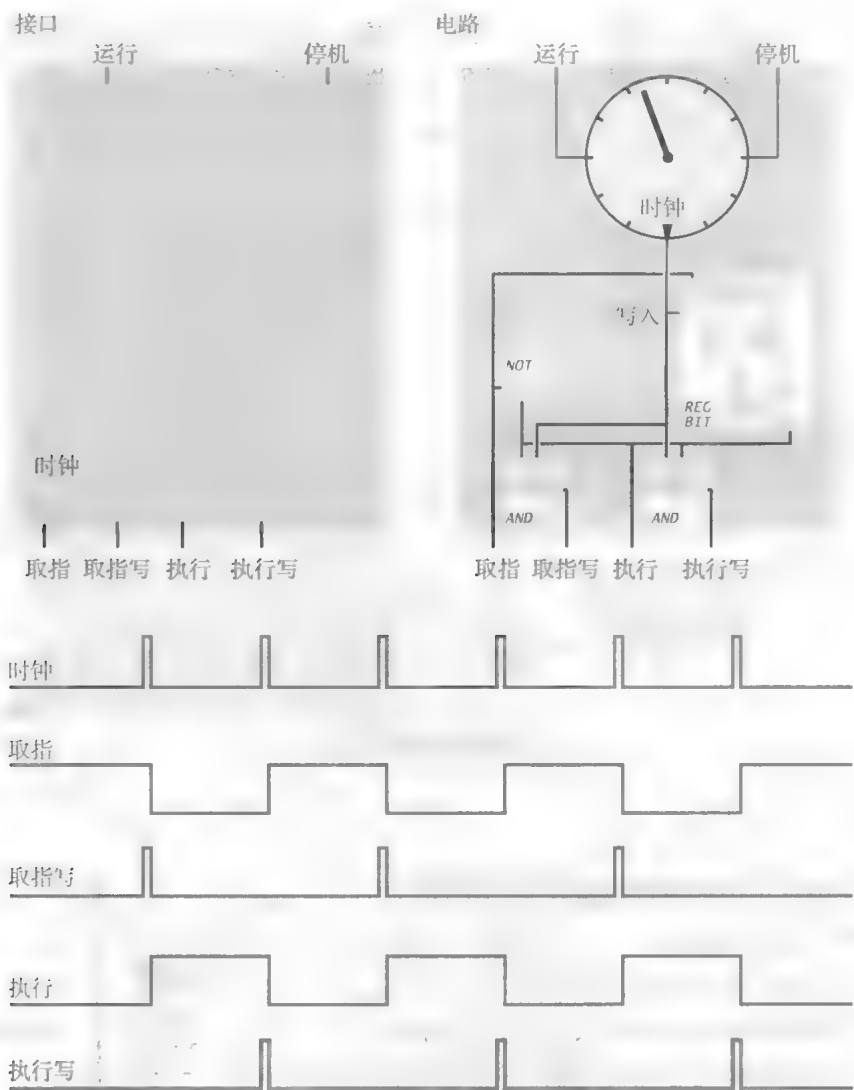
运行和停机输入。我们的时钟电路需要两个其他输入控制线，用来启动和停止时钟。运行 (run) 输入用来启动时钟——可以想象，它连接到计算机控制台上的“运行”按钮。停机 (halt) 输入用来停止时钟——当程序执行停机指令时，该输入被置为有效。

下图展现的是一个时钟电路，同时也考虑了刚才讨论的问题。我们计算电路中的所有控制线最终由四个时钟输出之一驱动。时钟电路控制输出会不断地进行如下循环：

- 取指令为 1，执行指令为 0。
- 取指写操作变为 1。
- 执行指令变为 1，取指令变为 0。
- 执行写操作变为 0。

正如你将在下一节中看到的，这个序列可以在计算电路中实现取指 - 执行循环，从而可以根据计算机的体系结构精确地改变寄存器和内存的状态。

时钟与我们对时序电路模块的研究非常接近，因为它充分说明了我们利用一个简单的电路和几个门来解决复杂行为的能力。同时，它还说明了对电路反馈进行严格控制的必要性，因为任何反馈都会导致不可预知的行为。而计算的本质是完全可预测性！



具有写入脉冲的取指 - 执行时钟

1061

**总结** 就组合电路而言，只要知道输入总线、输出总线和控制线的尺寸以及位置，我们就可以从更高的抽象层次上结合我们之前已经定义的模块来构建电路。这个过程再一次证明了抽象的力量：你不需要知道内部细节就可以完成电路的设计。

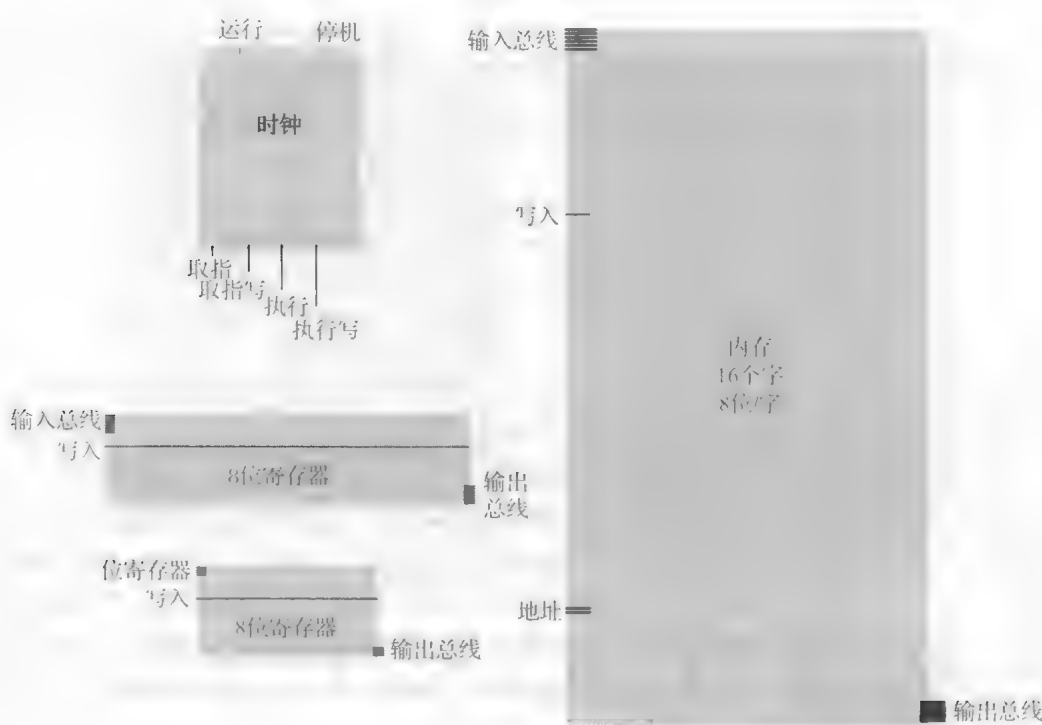
对于寄存器，只需要知道寄存器的内容总是在输出总线上可用，并且在写入控制信号有效时，可将输入总线上的值存储到寄存器中。

同样，对于内存，你只需要知道地址字的内容总是在输出总线上可用，并且写入控制信号的启动可将输入总线上的值存储到地址字中。

对于时钟，你只需要知道它可以产生我们已经讨论过的 4 个信号，无休止地重复驱动计算机的取指 - 执行周期的信号序列。

我们在计算设备中使用的时序电路模块如下图所示：一个 4 位寄存器、两个 8 位寄存器、一个 16 个 8 位字的内存和一个时钟。接下来，我们将研究如何将这些模块与上一节中的组合电路模块连接在一起（并添加少量的门电路）来制作计算设备。

1062



时序电路模块接口

1063

问答环节

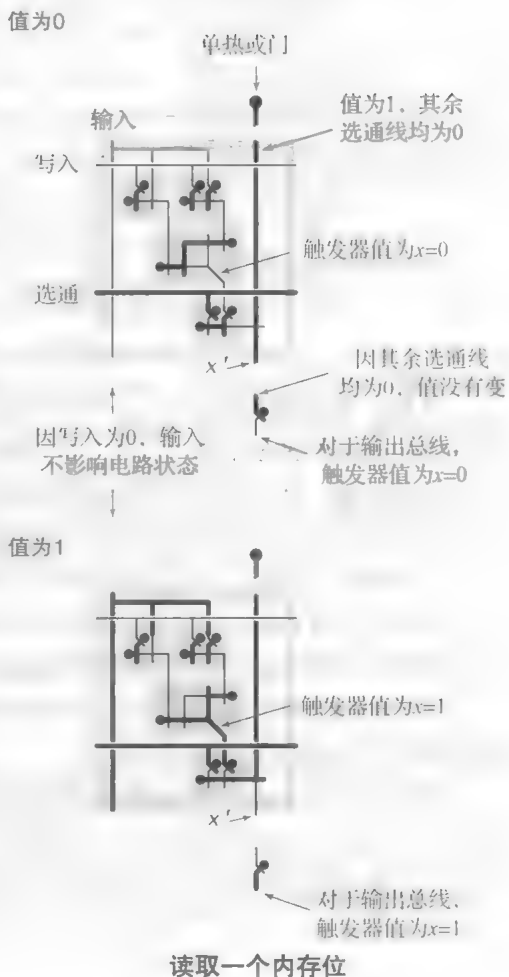
问：我对内存位值的读取机制还是不太明白，是否可以提供更多细节呢？

答：下图展示了详细的电路分析。结合多路复用器的概念或许更容易理解。如果你在之前的章节中已经了解解码器与多路复用器之间的区别，你就能看到在内存电路上的解码器、内存位的与门，以及每位的垂直或门，共同组成了一个多路复用器。

问：我有些不太明白时序图，这个很重要吗？

答：一方面，既然时序（timing）是一个重要概念并在现实技术中有不同的表现方式，那么我们就需要认真学习并严谨对待。另一方面，重复阅读我们如何设置一个寄存器位的值（要确保我们理解写入控制线的操作），以及如何设置时钟电路（要确保我们理解它实现的控制线序列），这是值得花费时间的。幸运的是，我们不需要比这些更复杂的概念——从现在开始，我们可以只需要在模块级别分析并构建计算机。

1064



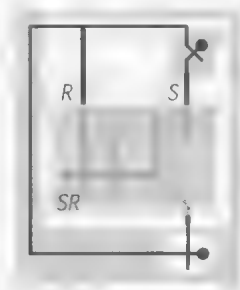
读取一个内存位

练习

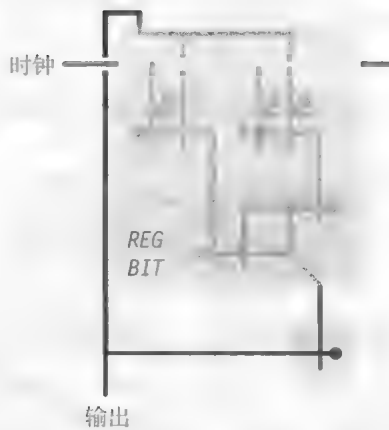
7.4.1 利用一对交叉耦合的或非门设计一个触发器。  
答案：(使用的材料稍多于正文中的解决方案)。



7.4.2 描述以下电路实现什么？它是否稳定？



7.4.3 描述以下电路的行为。



答案：输出在 0 和 1 之间交替，每一次时钟脉冲变化一次。

- 7.4.4 设计一个使用了三个开关的蜂鸣器（一种不稳定反馈的电路）。  
7.4.5 使用 8 个内存位和一个解码器 / 多路分配器，设计一个具有 4 个内存字、每个字 2 位的内存。  
7.4.6 使用 8 个内存位和一个解码器 / 多路分配器，设计一个具有 2 个内存字、每个字 4 位的内存。  
7.4.7 为具有  $2^m$  个  $n$  位内存字的内存中门的数量设计一个公式，使用这一公式验证正文中“不同规格的内存中门的数量”表格。

1065

1066

创新练习

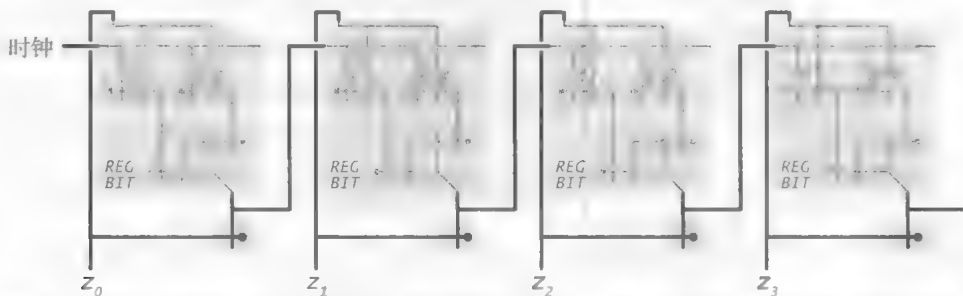
- 7.4.8 电路分析。说明以下每个电路使用的开关数量。  
a. 一个寄存器位  
b. 一个内存位



- c. 一个 8 位寄存器
- d. 16 个 8 位内存
- e. 一个时钟电路

计数每个开关（比如，一个 2 路与门有 4 个开关）。

- 7.4.9 设计一个寄存器。编写一个 Java 程序来实现一个  $n$  位寄存器，从命令行中读取一个整数作为参数  $n$ 。
- 7.4.10 设计一个寄存器（接上题）。在上一个练习编写的程序基础上继续修改，使程序读取第二个命令行参数  $x$ ，并使得设计出的寄存器中存储  $x$  的二进制形式（写入信号为 1）。若有第三个命令行  $y$ ，设计一个寄存器使其写入信号为 0，并且其输入线上的数据为  $y$  的二进制形式。
- 7.4.11 设计一个内存。编写一个 Java 程序，从命令行中读取整数  $m$  和  $n$ ，实现一个具有  $2^m$  个  $n$  位内存字的内存。
- 7.4.12 设计一个内存（接上题）。从标准输入中读取  $2^m$  个整数，在上一个练习题编写的程序的基础上，实现一个存储这些整数的内存，然后添加功能，以实现新值写入内存的过程。
- 7.4.13 双端口内存。设计一个内存模块，具有 2 个输出总线、2 个地址线、2 个读取控制线，在输出总线上可以输出 2 个内存字的内容。
- 7.4.14 时序。当写入脉冲输出到一个  $2^m$  个  $n$  位内存字的内存上时，给出被激活的开关链条的最大长度。
- 7.4.15 时钟。描述如何建立一个时钟电路  $(m, n)$ ，电路详细说明如下：假设所有开关操作均需要一个固定时间  $x$  秒，当导线连接到电源时，导线上所有的值瞬时变为 1。一个  $(m, n)$  时钟电路在  $mx$  秒内生成信号 0，在  $nx$  秒内生成信号 1，并以  $(m + n)x$  秒为一个周期，在两个输出值之间不断转换。
- 1067 7.4.16 二进制计数器。描述如下电路在输入线上产生一系列时钟脉冲时的行为。



答案：这是一个计数器（counter）。如果将  $z_3z_2z_1z_0$  看作一个二进制值，那么每个时钟脉冲后这个值会增 1。在练习 7.4.3 的基础上理解这个电路。

- 7.4.17 循环计数器。用重置（reset）、写入（write）控制输入，以及输出  $z_0 \sim z_5$  作为 6 个寄存器位来设计一个电路，该电路可以满足：reset 将信号  $z_0$  设置为 1，其他触发器设置为 0，写入脉冲的值依次在：100000, 010000, 001000, 000100, 000010, 000001, 100000, ... 之间循环。
- 7.4.18 LFSR。设计一个电路来实现线性反馈移位寄存器（LFSR），如练习 7.4.15 中所示。用 12 个寄存器位来构建电路，从右到左依次标记为 0~11，以位 0 作为输出，输入来自输入总线。将一个时钟信号作为控制输入。每次时钟脉冲开始，用位 11 的值和位 9 的值进行异或运算后送到位 0 中，将其余各位的值依次向左移动一位，而位 11 的值忽略。这个设备可以加载一个 11 位的数据作为种子，并在每次时钟滴答后产生一个“随机”位。

7.5 数字设备

作为最后一节，本节主要解决的问题是，连接和控制我们在前两节中所讨论的电路模块。

第一个要处理的是计算部分，第二个是存储部分，第三个是控制部分（control）。ALU、总线复用器、寄存器和存储器占 CPU 门电路总数的 99% 以上，从某种意义上来说，我们已经基本完成了。但从另一个角度来说，我们刚刚开始能够解决“计算机如何运行”的问题，因为剩余的门电路将解决的是控制信息如何以及何时传给处理器。它们解决的是计算机核心的问题，即“执行程序”的问题。

TOY-8 首先，我们明确一下我们的目标：为我们的 TOY 系列最小的虚拟计算机设计一个 CPU。我们之前已经提及了关于 TOY-8 的许多信息，现在我们将细化这些问题。

基础参数。我们真正的目标是易于讲解：我们想在本书的两页纸上展示一个完整计算机的电路实现，可以看到每个开关。因为 TOY-8 是一个非常小的计算机，只有 16 个 8 位字的存储器和一个寄存器。尽管有这些或多或少的限制，但用 TOY-8 完成非常重要的计算任务也不是不可能的（但我们不会花时间这样做）。

指令系统。由于只有一个寄存器，TOY-8 的指令格式与 TOY 格式有很大不同。具体而言：

- 每条指令都隐含地指向唯一的寄存器，所以指令不需要特别地指向寄存器。
- 为了指定一个内存字的地址，我们需要 4 位二进制数。
- 为了指定一个操作码，我们需要 3 位二进制数。
- 位 0 未被使用（一直保持为 0）。

TOY-8 指令的格式如下图所示。



TOY-8 指令解析

这些规定体现了指令集体系结构中涉及的因素，这是早期计算机设计中相当重要的一个因素。例如，我们可能并没有确定在计算机的下一个版本中是否要支持 16 条指令或 32 个字的内存。现在你可能会明白早期的设计师为什么留下一些未使用的位，这样可以在后续修改时保持一些灵活性。我们这里的设计目

1070

标是尽可能简单地开发一台能够说明真实计算机主要特性的计算机，所以我们认为 8 个指令和 16 个字的内存足以满足我们的需要。

完整的 TOY-8 指令集在右表中给出。它缺少在 TOY 指令集中的减法 (subtract)、移位、间接寻址，以及在 TOY 指令集里的三种不同的分支/跳转指令，但它还是实现了算术、条件和循环，能够满足计算的基础需要。

| 操作码  | hex | 描述      | 伪代码                    |
|------|-----|---------|------------------------|
| 0000 | 0   | 停机      |                        |
| 0010 | 2   | 加法      | $R = R + M[addr]$      |
| 0100 | 4   | 按位与     | $R = R \& M[addr]$     |
| 0110 | 6   | 按位异或    | $R = R \wedge M[addr]$ |
| 1000 | 8   | 加载地址    | $R = addr$             |
| 1010 | A   | 加载      | $R = M[addr]$          |
| 1100 | C   | 存储      | $M[addr] = R$          |
| 1110 | E   | 为 0 则跳转 | if (R == 0) PC = addr  |

TOY-8 指令集

由于未使用的位始终为 0，因此我们使用两个十六进制数来描述指令时，操作码总是偶数。例如，指令 2E 是“将寄存器 R 的内容加上存储单元 E 的内容再赋给 R 寄存器”，指令 CE 是“将 R 的内容存储到存储单元 E”。

与 TOY 一样，假设存储单元 F 连接到标准输入 / 输出。我们还假设内存地址 0 始终存储 0。TOY-8 的程序。举例来说，以下是利用 TOY-8 指令实现的程序 6.3.3，对输入的数进行求和：

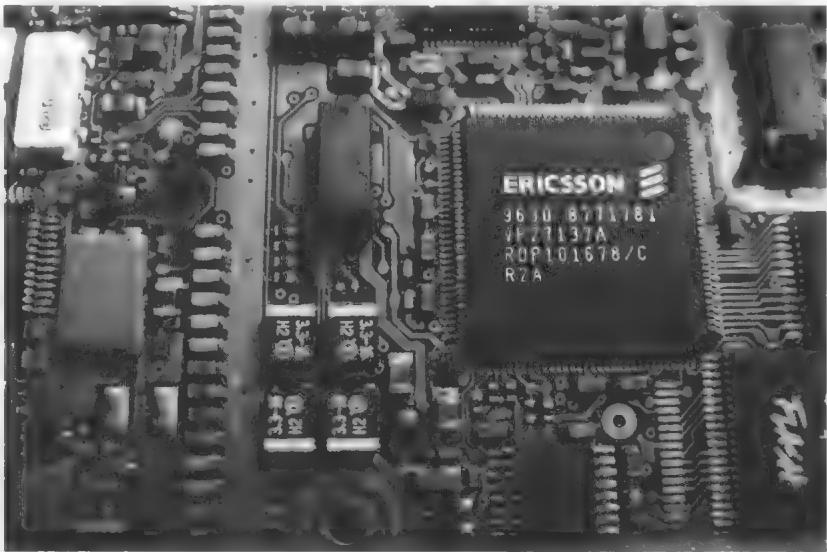
1071

```
1 A0 R = 0
2 CE M[E] = R int sum = 0
3 AF R = stdin while (!StdIn.isEmpty())
4 E9 if (R == 0) PC = 9 {
5 2E R = R + M[E] c = StdIn.readInt()
6 CE M[E] = R if (c == 0) break
7 A0 R = 0 sum = sum + c
8 E3 if (R == 0) PC = 3 }
9 AE R = M[E]
A CF stdout = R StdOut.println(sum)
B 00 halt
```

请注意，寄存器（R）的使用率极高。例如，为了做一个无条件的跳转，我们需要首先将 0 加载到 R 中，然后再执行 branch 指令跳转到地址 0。编写这种代码的经验恰恰是早期计算机设计者决定设置多个寄存器的原因。决定采用 32 个字的存储还是 16 个指令，确实很难。当然，从设计的角度来看，增加更多的内存是很容易的，但事实上每扩展一位的存储也势必面临着成本增加的挑战。同样，增加更多的指令需要额外的电路设计，也是一个挑战。

所有这些细节都比较重要，因为我们目前的重点并不是使用 TOY-8 编程，而是开发一个实现 TOY-8 的电路。这个讨论之所以有价值，是因为 TOY-8 与 TOY 和真机相似，除了规模之外。通过我们增加更多的内存或者更多的指令，我们可以用 TOY-8 这样的机器来实现前面章节中所有的程序，而这会使我们置身于一个 TOY 的计算世界，许多基本特性就像现在真实的计算机上的一样。

我们不考虑计算机的物理特性，如键盘、显示屏、电池、电源连接等，我们的兴趣在于计算电路的设计——CPU。打开计算机的电路板，你会发现一个包含这个电路的黑色方形“芯片”。接下来，我们来了解这个芯片内部的问题。



中央处理器

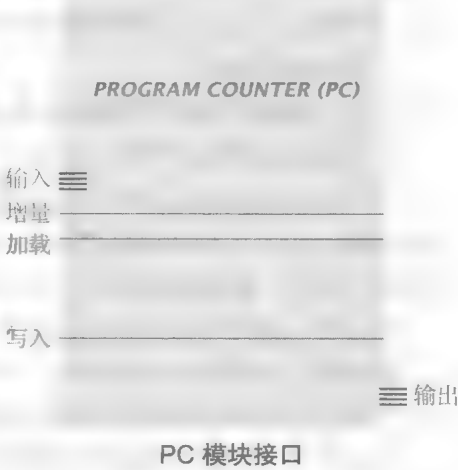
热身 作为设计一个完整计算机的热身运动，我们先从计算机数字设备的核心开始：程序计数器（PC）。回想第 6 章，程序计数器的目的是跟踪当前正在执行的指令的地址。其值可以通过以下两种方式之一进行更改：或者自增（大部分时间），或者赋给一个全新的值（对于分支条件）。

程序计数器说明了我们在实现数字设备时需要考虑的重要因素。具体而言，我们需要解决以下问题：

- 我们需要哪些模块？
- 信息如何从一个模块传播到另一个模块？
- 控制线路有哪些？控制信号的时序如何？

在构建寄存器和内存模块时，我们已经考虑了这些问题。现在，我们面对的是一个更为复杂的数字设备——PC，因此需要再一次综合考虑以上问题。后文的图展示了最终结果。

接口。PC 的主要功能是保存当前指令的地址，所以我们期望地址始终在输出总线上可用。PC 的值可以通过两种方式进行更改：或者递增，或者更改为不同的值（用于分支指令）。因此，我们需要两个控制信号（自增和加载）和一个跳转地址的总线输入。我们还需要一根写控制线来控制 PC 值的在何时被修改。综合以上分析，得出如右图所示接口。



模块。我们至少需要两个模块：

- 一个寄存器来保存地址。
- 一个增量器，可以在原来地址的基础上加 1。

以上仅仅是构建计算机数字设备的开端。后面会看到，当我们考虑模块交互时，还会需要另一个模块。

总线连接。为了将信息从一个模块传输到另一个模块，我们需要用导线在模块间建立连接。大多数情况下这样的导线都是总线连接（bus connection）的一部分，通常将一个模块的输出总线连接到另一个模块的输入总线上，每一位用一根导线连接。鉴于我们有一个寄存器和一个增量器，我们至少需要以下四条总线连接：

1073

- PC 寄存器到增量器（要自增的数字，即当前地址）
- 增量器到 PC 寄存器（自增后的结果）
- PC 输入到 PC 寄存器（新值，在跳转时使用）
- PC 寄存器到 PC 输出（用于取指令）

以上描述都是缩略的写法，因为我们的总线连接总是将一个模块的输出总线连接到另一个模块的输入总线。因此，“PC 寄存器到增量器”实际上的意思是“PC 寄存器输出总线到增量输入总线”。

从这个列表中我们可以看到 PC 寄存器有两个输入和两个输出，但是输入和输出之间存在不对称。我们可以拆分一个模块的输出来连接两个不同的输入，只需要 T 形连接——拆分之后两个总线之间的值完全相同。但是我们不能将来自两个不同模块的输出结合到一个模块的输入端，因为这些线可能会有不同的值——我们必须使用一个开关来隔离它们（通常是总线多路复用器）。相应地，我们添加一个 2 路的总线复用器，用来代替前面列出的第二个和第三个总线连接：

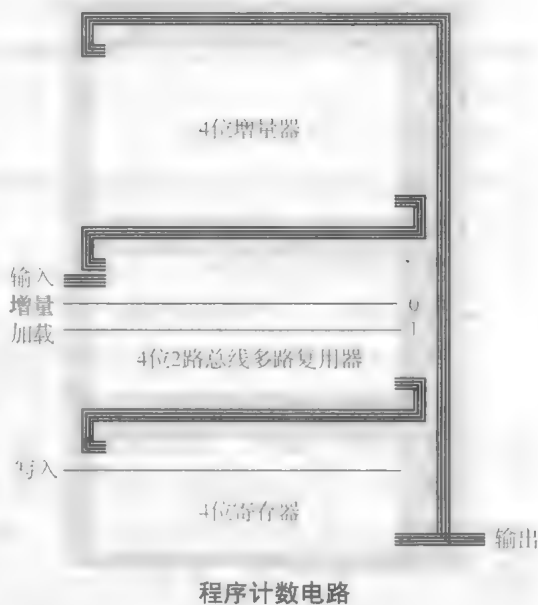
- 增量器到总线复用器 0。
- PC 输入到总线复用器 1。
- 总线复用器到 PC 寄存器。

综合上述设计，得到的线路图如右图所示。

为了清楚起见，我们遵从输出总线从模块右下方输出，输入总线从左上角输入的惯例。有时候，从模块的同一侧连接可以使得导线的路径更短，但是这样会难以区分输入和输出，并且有时需要翻转扭曲总线来保证位的顺序。

控制线。控制线直接与我们使用的模块相连接，并精确对应于我们设计的接口上的控制线：

- 控制总线多路复用器的是两根线（自增和加载），以允许我们选择 PC 是自增还是加载新地址。
- 写入控制线用于控制寄存器从总线上加载数据的过程。



1074

与其他总线多路复用器一样，我们不会将自增和加载控制信号同时设置为 1。与存储器电路一样，在写入脉冲到来之前不会进行任何操作。如果此时自增为 1（且加载为 0），则寄存器进行自增；如果此时加载为 1（且自增为 0），则 PC 输入总线上的值将加载到寄存器中。

请注意，此电路的总线连接有很长的周期。当自增信号为 1 时，如果此时没有写入控制线，电路将无限循环，增加 PC。若有写入控制线，我们可以确保 PC 与其他模块进行同步改变。我们在计算机电路设计的过程中会反复用到这一机制。

连接和时序。PC（甚至任何数字设备）的行为完全取决于得到的控制信号和输入数据，以及它们到来的顺序，而这些都来自于其他模块的连接。我们的计算机有四个这样的连接：输入总线、自增控制线、加载控制线和写入控制线。

- 输入总线的值由 IR 中的地址位决定的（当正在执行的指令是跳转时）。
- 自增或者加载控制值（只能是二者中的一个）由在执行阶段的时钟周期内决定的，取决于当前指令是否为跳转。如果当前指令是跳转且 R 为 0 则为加载，否则就是自增。
- 写入控制取决于来自时钟周期的执行写入信号，这将导致新值存储到 PC 寄存器中。

通过这些连接，来自时钟的信号周期决定了电路的行为：不管是增量值还是跳转地址都会在执行周期结束时加载进 PC 寄存器。PC 的内容总是在输出总线上可用（但是它只在取指阶段使用，提供下一条取指令的地址）。

总之，我们的 PC 有 3 个模块、3 根内部总线连接、1 根输入总线和 1 根输出总线，以及 3 个控制输入。这对我们整体理解 CPU 会是一个很好的热身，因为它有助于理解取指-执行周期中时钟信号的作用，以及写使能信号在控制和同步内存位状态变化中的作用。如果你了解 PC 的工作原理，那么你也将会很容易理解 CPU 的工作原理。

如同之前的思路，现在我们可以更高的抽象层面应用 PC 电路。因此，在后续的设计中，我们会保留原来的接口，并且知道只要提供了相应的控制信号，PC 相应的控制值将做出相应的改变。

1075

**TOY-8 CPU 的组织 and 连接** 最后，我们准备实现本章的目标：设计一个完整的 CPU。

我们遵循与 PC 相同的方法，但会有更多的模块、总线连接和控制线。值得注意的是，实际上没有那么多：PC 有 2 个模块、1 个总线多路复用器、5 根总线连接和 3 根控制线，而 CPU 只有 7 个模块、2 个总线多路复用器、10 根总线连接和 16 根控制线。

运行

CPU 接口

接口。我们的 CPU 电路直接与外部相连，而不是另一个内部的电路模块。例如，TOY-8 面板上的 RUN 按钮用于启动时钟，就会与 CPU 的 RUN 信号相连接。除此之外，我们省略了这些连接的细节：连接到开关、按钮和前面面板上的指示灯的线，以及与 I/O 设备的连接。为了构建 CPU，我们可以假定内存和 PC 保存着通过硬件提供的初始值（程序及其起始地址）。现在，这个基本功能已经从程序员利用开关发送二进制代码发展到在时钟开始之前利用专用的硬件来初始化整个存储器。另外，从 CPU 的角度来看，标准 I/O 设备是纸带读取器 / 打孔器还是连接到互联网也没有区别，因此我们也不需要考虑该接口的细节。

模块。我们的 TOY-8 CPU 电路由七个模块组成。除了下面这个列表中的最后一项以外，所有这些都很熟悉，因为自从第一次介绍 TOY 后，我们已经多次提到它们。

- ALU。
- 处理器寄存器 (R)。
- 指令寄存器 (IR)。
- 程序计数器 (PC)。
- 主存。
- 取指 - 执行时钟 (写入脉冲)。
- 一个称为 CONTROL 的组合电路，用于管理控制线路。

本章前半部分未介绍到 CONTROL，因此 CONTROL 的设计和实现是本节的重点。

1076

总线连接。大多数总线用于数据的传输，它们由 TOY-8 中的 8 条线组成；还有一部分用于地址位的传输，它们由 3 根导线组成。在一个 64 位机中，总线的宽度是设计的主要考虑因素，为了传输一个内存字，总线需要 64 根线。即使在 TOY-8 中，你也会看到总线设计是电路的一个重点。

机器内的总线连接由指令集的需求决定。例如，要在 TOY-8 机器上执行存储指令，IR 的地址位必须连接到存储器的地址线，而 R 必须连接到存储器的输入总线。以同样的道理分析 TOY-8 中的所有总线连接，如以下表格所示。

| 指令      | 总线连接                                    | 指令    | 总线连接              |
|---------|-----------------------------------------|-------|-------------------|
| 所有指令的取指 | PC到内存的地址线<br>内存到IR                      | 地址加载  | IR地址到R            |
| 停机      | 无                                       | 加载    | IR地址到内存地址<br>内存到R |
| 加法、异或、与 | IR地址到内存地址<br>内存到ALU0<br>R到ALU1<br>ALU到R | 存储    | IR地址到内存地址<br>R到内存 |
|         |                                         | 为0则跳转 | IR地址到PC           |

TOY-8 指令的总线连接

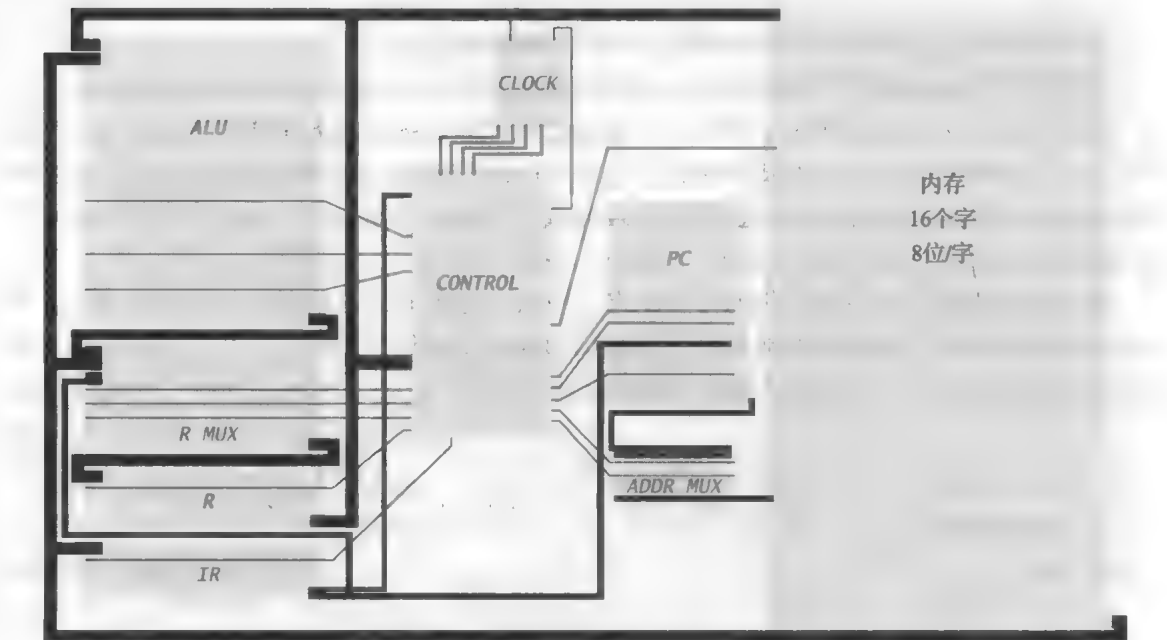
正如我们在 PC 上看到的，我们不能有两条不同的输出总线连接到同一条输入总线上，所以这个连接列表意味着需要两个总线多路复用器：一个 3 路总线多路复用器用于输入到 R 的转换（ALU、IR 地址位和内存）以及一个 2 路总线多路复用器，用于转换内存的输入地址总线（PC 和 IR 地址）。我们分别称之为“R 复用器”和“MA 复用器”。其他的连接都是从一个模块的输出总线直接连接到另一个模块的输入总线。

后面表格展示了模块、总线多路复用器和 TOY-8 CPU 的总线连接（以及稍后会介绍到的控制线）。为了更好地理解总线连接表，请仔细看参考图。控制线。我们现在的任务是组织控制我们的模块和多路复用器的线路。每条控制线都是模块的输入，并选择执行的指令。正如左表所示。所有的控制线由时钟信号驱动，所以每个控制线都需要连接到一个时钟信号。而 CONTROL 电路的功能就是对这些连接进行管理，我们将在本节后面介绍。在这之前，我们按照不同的指令罗列所有需要的控制线，每四个时钟信号一条指令，并汇总在后面的表格中。接下来，我们会更详细地描述。

取指令。首先，考虑 CPU 的取指令阶段，这很简单，因为所有指令的控制信号都是相同的。取指令的目的是按照 PC 中的地址将指令加载到 IR 中。这个行动分两步完成：

- 取指令时钟信号将地址多路复用器（address mux）PC 控制线设置为 1。
- 取指令写时钟脉冲直接连接到 IR 写入。

由于 IR 唯一的总线输入是来自内存的输出，所以这个时钟序列将导致所寻址的字（下一条指令）被加载到 IR 中。



TOY-8 CPU 的布局、总线连接和控制连线

执行。对于执行阶段，控制信号序列依赖于当前的指令。的确，每个控制信号序列实现了每条指令的功能。大多数指令为 R 写入一个新的值，所以对于 store 指令，执行写入时钟



脉冲驱动内存进行写回；对于算术指令、load 指令、load addr 指令，驱动 R write 以写入寄存器 R。在其他情况下，控制信号用于控制总线复用器的开关，或者用于为 ALU 选择适当的操作。

另外，在执行阶段，PC 的值总是在变化，因此执行写入时钟信号连接到 PC 写入控制线。对于除了跳转指令（branch if zero）之外的指令，执行时钟信号驱动 PC 会自增，如果执行跳转指令，则执行时钟将跳转地址写入 PC。

为了更好地理解这个表格，我们会仔细讲解每条指令，并依次回答以下问题：指令的功能是什么？总线如何连接以完成其目的？哪一个控制信号序列可以完成这项工作？我们依次来看。

停机。停机指令使得 CLOCK 停止控制线变高，从而停止时钟。

算术指令。随着总线连接的建立，以及 MA 多路复用 IR 控制信号变为高电平，ALU 会计算加法、按位异或、按位与等，操作数分别为 IR 和 R 所指向的操作数，但是执行时，只有相应控制信号为 1 的结果才会出现在 ALU 的输出总线上。该总线连接到 R 的多路复用器，因此在执行期间，当 R 写入脉冲上升时，其线上的值被存储在 R 中。

加载地址（load addr）。在执行过程中 R 多路复用 ALU 控制线变为高电平，在执行写入期间，借助 R 写入（R write）信号，将来自 IR 的地址位存储在 R 中。

加载（load）。在执行过程中，R 多路复用 IR 控制线和 MA 多路复用 IR 控制线的高电平到来，在执行写入期间的 R 写入（R write）会使得所寻址的内存字被存储在 R 上（由于存储器输出总线连接到 R 的多路复用器）。

存储（store）。在执行过程中 MA 多路复用 IR 控制线变为高电平时，在执行写入期间（由于 R 的输出总线连接到内存的输入总线），R 写入（R write）会使得所寻址的内存字被存储在 R 上。

PC 自增。在执行期间 PC 自增控制高电平时，在执行写入期间由增量器计算的值被存储在 PC 中，依据执行写入期间的 PC 写入，除非当前指令是转移指令（branch if zero），且 R 为 0。

PC 加载。如果当前指令是转移指令，且 R 也为 0，在执行过程中 PC 加载控制高电平时，在执行写入期间，PC 写入会将 IR 地址线中的值存储在 PC 中。

总之，每条指令都是由一系列控制信号来实现的。每个序列由时钟决定，以响应另一个组合电路 CONTROL，我们接下来讨论 CONTROL 电路的实现。

CONTROL 电路 后文展示了将时钟信号传递给控制线的 CONTROL 电路的设计图。值得注意的是，它由两个复用器和五个门组成。时钟电路会产生一个无限循环：取指令（fetch）、取指写入（fetch write）、执行（execute）、执行写入（execute write），它们出现在电路图的顶部。接下来，我们详细讨论对这些时钟信号序列的响应。当然，理解这个响应

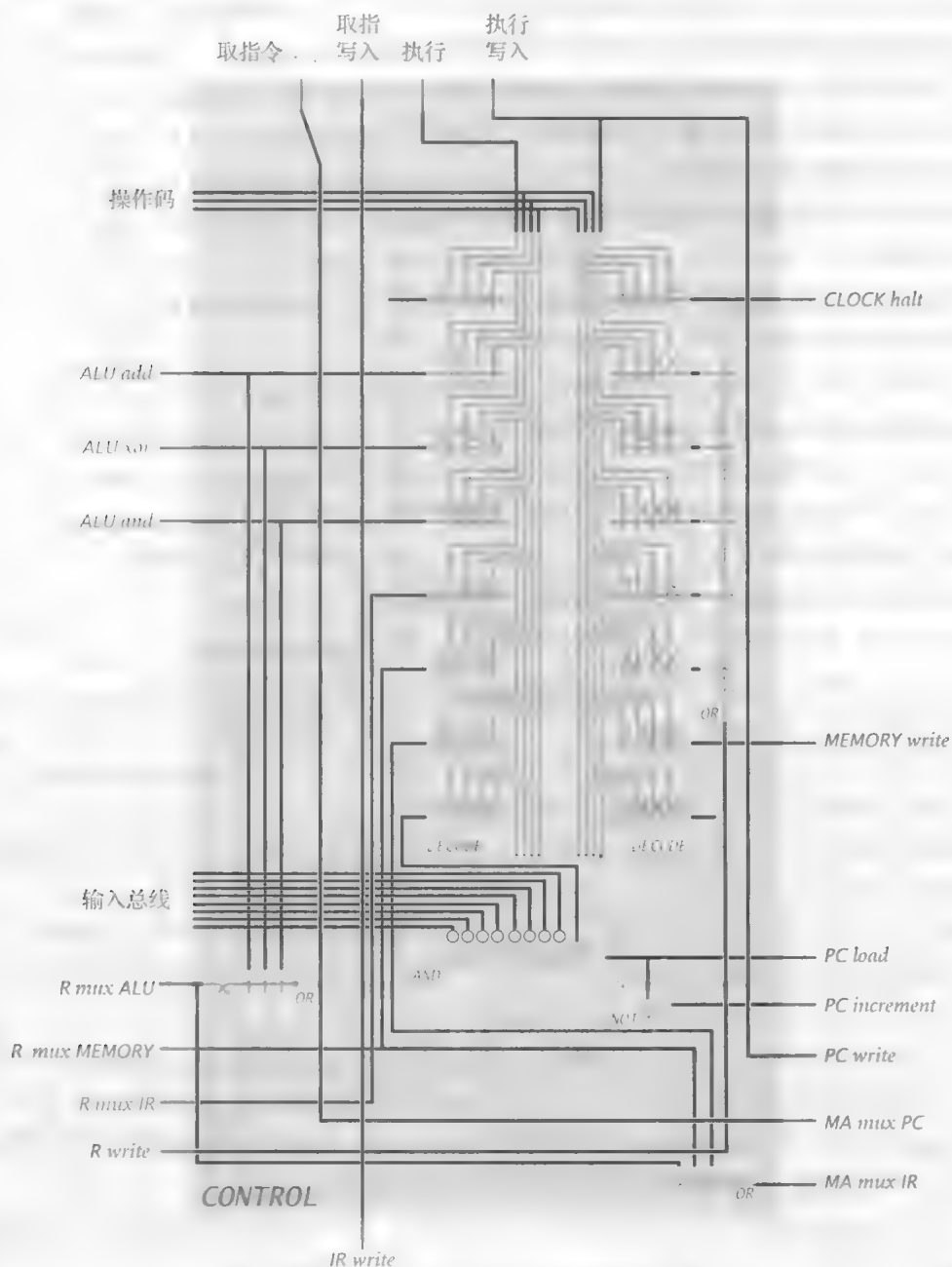
| 指令                          | 取指                   | 取指写入    |
|-----------------------------|----------------------|---------|
| 所有指令                        | address mux PC       | IR      |
|                             | 执行                   | 执行写入    |
| 加法                          | MA mux IR            |         |
|                             | ALU add              | R       |
|                             | R mux ALU            |         |
| 停机                          | 停机                   |         |
| 异或                          | MA mux IR            |         |
|                             | ALU xor              | R       |
|                             | R mux (ALU)          |         |
| 与                           | MA mux IR            |         |
|                             | ALU and              | R       |
|                             | R mux (ALU)          |         |
| 加载地址                        | R mux (IR)           | R       |
| 加载                          | address mux (IR)     | R       |
|                             | R mux (memory)       |         |
| 存储                          | address mux (IR)     | 内存      |
| 除了 branch if zero 成功之外的所有情况 | PC increment         | PC      |
|                             | branch if zero 成功的情况 | PC load |

TOY-8 结构的控制线

的关键是要认识到它非常依赖于机器的当前状态，特别是指令寄存器中操作码位的值。这个电路对模块输入/输出约定有一个例外：它的输入在顶部，而它的输出分布在另外三个边上。

取指令。对取指令的响应很简单：线路直接选通到 MA 多路复用器 PC 控制线，指示该多路复用器将 PC 中的地址传递到内存作为地址输入。

1080



1081

基于 TOY-8 CPU 的控制时序的组合电路

取指写入。对下一个取指写入脉冲的响应也很简单：线路直接连接到 IR 写（IR write）控制线。结合取指令（刚刚描述），可以将 PC 中的地址指向的内存字加载到 IR 中。

执行。CONTROL 中间左侧的复用器是一个多路选择器（demultiplexer），它将执行信号

的值放在八个输出行之一上，具体哪行由操作码指定。因此电路的行为完全由操作码决定。例如，如果操作码指定一个异或操作，则多路选择器将激活往下数第三行，从而激活 ALU 的异或控制线以及 R 复用器的 ALU 选通。你可以通过激活你所预期的控制线来更好地了解该电路，并参考上一小节中的表格。跳转指令（branch if zero）具有特殊性，即若 R 上的所有位为 0，则跳转会生成 PC 上的加载控制信号。若指令不是跳转或者 R 上任意一位不是 0，非门就会生成 PC 的增量控制。

执行写入。图中右侧的多路分配器是一个开关，它会把执行写入信号的值传输到八个输出线之一，具体哪个线由操作码指定。同样，电路的行为因此完全由操作码决定。复用器输出分成了多路，分别对应 add、xor、and、load address 和 load，这些信号最后进入一个或门，激活通用寄存器 R 的写信号，使得适当的值存储到 R 中；store 指令针对的多路复用器输出直接激活内存写入（memory write）信号，使得通用寄存器 R 的内容被存储到内存中。执行写入信号的写入使能信号也直接连接到 PC 写入信号，因此脉冲会同时将新值加载到 PC 中（或者增加或者从 IR 地址线中加载，即自增或加载）。

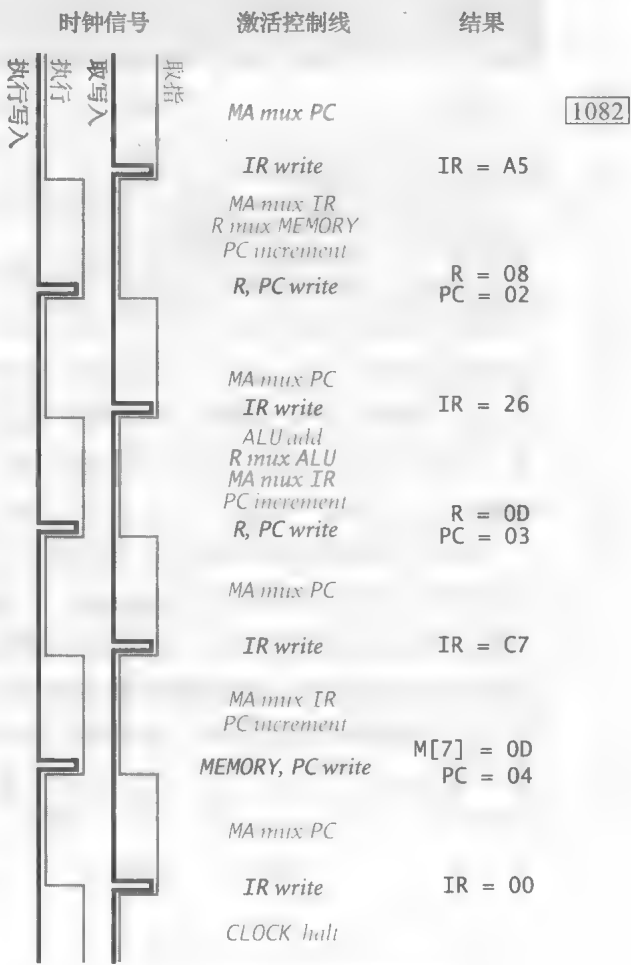
举例：一个 TOY-8 程序 作为最后一个例子，我们讨论一段 TOY-8 的系列控制信号，它的作用将等价于“你的第一个 TOY 程序”，也就是两个数字求和：

```
1 A5 R = M[5]
2 26 R = R + M[6]
3 C7 M[7] = R
4 00 halt
5 08
6 05
7 00
```

我们假设内存空间的 1 到 7 位置上已经加载好了这些数字，而 PC 被设置为 01。可以想象一下，程序员可能是使用开关和按钮（如 6.2 节所述）传输的数据，也可能是通过计算机内的专用初始化硬件来完成这些设置，最终结果都是内存和 PC 的起始状态被设置成了预想的状态。我们感兴趣的是时钟开始后发生的事情。

在右图中，由时钟产生了四个信号的循环——取指令，取指写入，执行，执行写入。只要时钟还在工作，就会在这四个信号之间不断重复。中间一栏列出了由时钟信号激活的控制线序列，右边一栏展示了内存、IR 和 PC 状态的变化。仔细研究这个图，你将理解本节的核心思想：任何 TOY-8 程序由一系列周期性的时钟信号引起的控制线的激活来实现。值得一提的是，简单的 CONTROL 电路可以通过这种方式实现 TOY-8 的全部指令集。

“你的计算机就是这样工作的……”。一个小电路将一个周期性的时钟脉冲转换成一个无



一个 TOY-8 程序的控制线激活序列

限循环的时钟信号，这个循环引起一系列控制信号的激活，以改变机器的状态——当然，状态的变化是相对应机器当前的状态而言的，当前的状态是由 PC（执行指令的地址）和 IR（指令本身，特别是指令操作码）所决定的。如果现有再有人问起计算机的操作原理，你应该可以解释了。

1083

**展望** 在后面，你将看到一个我们所描述的 TOY-8 CPU 的完整电路，详细到可以从底层开关级来分析电路。到此，我们实现了本书中阐述的主要目标之一——了解计算机的内部工作原理。在此，还有几个要点值得回顾。

正如我们强调的那样，TOY-8 与计算机的主要区别在于规模，而规模大小的差异很容易改进。例如，我们可以轻松地扩展设计，将它扩展成一个 32 位计算机，配有 29 位地址和  $2^{29}$  个（超过 5 亿）32 位字的内存——当然这会需要超过 160 亿个触发器，而现在的 TOY-8 只需要约 150 个触发器。相比之下，将指令数量加倍可能需要一些复杂的逻辑，但是也可能只是控制电路翻倍，相比之下可以忽略不计。这里存在一些夸大的成分，因为 TOY-8 的 ALU 仅实现了三个基本的运算操作，现代 CPU 设计中注重扩展多种操作，包括浮点数和内存管理等。但是，我们还是要说明，这样的设计只会使系统的复杂度随着字宽的增加而线性增长，而内存的增加则会呈指数级增长。

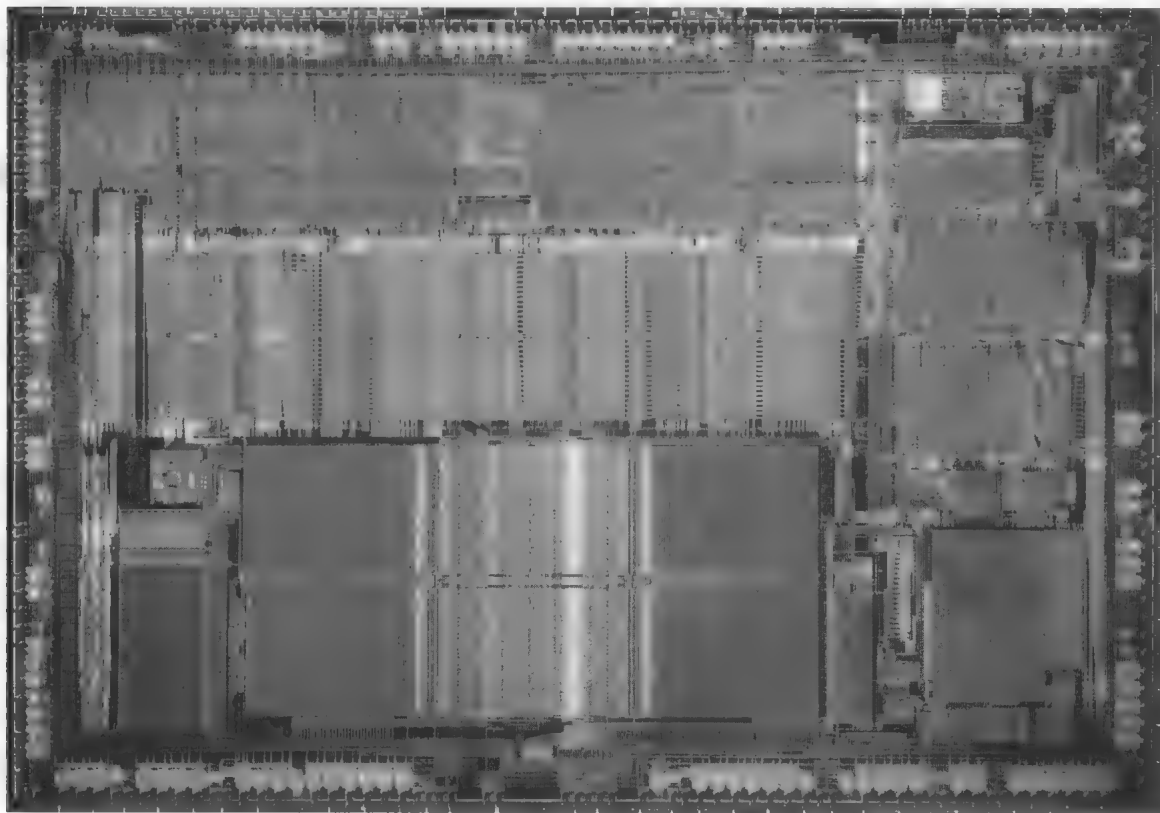
正如我们以前强调的那样，TOY-8 和你的计算机都是冯·诺依曼机器——指令和数据之间没有区别。当然我们可以在构建电路时刻意加以区分，可以想象一个电路由两个不同的存储器组成，一个用于指令，一个用于数据。事实上，出于安全原因，人们今天确实是这样设计电路的。

布局在我们的设计中起着核心作用。如前所述，数字电路的抽象世界与真实的物质世界之间有着密切的联系。现在人们利用计算机程序来绘制他们的计算电路，然后将这些绘图发送到制造芯片的计算机驱动系统。

**警告：**本章的目标是帮助你了解计算机的工作原理，而不是教你如何建立最先进的高性能电路。我们所做的每个设计选择都是为了使电路简单易懂。例如，我们可能会通过旋转模块以节省设计空间。再如，设计中无须体现每一个开关：如果你设计真正的计算机，你将会用到门（抽象层次更高的开关），通常也会用到布局。不过，我们的要点是：你可以设计出你自己的计算机，你可以编写一个程序来扩展这个计算机的规模，你可以假想它能够被构建出来。

1084

如果你要拿出显微镜，以 10 万倍左右的倍数观察计算机上的 CPU 芯片（可以肯定的是，现在对你来说可能并不那么容易），你会发现它与抽象的设计看起来没有太大的区别。当然，真正的计算机会受到各种各样的物理约束，设计也会更加复杂，使用到的模块类型也比我们学到的多，因此你可能需要对设计过程了解更多才能看懂这些细节。即使是这样，你还是可以认出总线、寄存器、内存和其他模块，也可以看到每个开关。相应地，对现实条件进行一些妥协后，我们可以依据下面图示中的设计，实际制造一个物理的 TOY-8 CPU。



Intel 80486 微型处理器的芯片内部图片

所有的计算基于两个简单的抽象——开关和时钟，这是一个放之四海皆准的基本设计理念。为了使计算机拥有更多的内存和寄存器，我们需要更小、更快的开关和更快的时钟。与从头开始构建新设计相比，充分利用微小的技术改进来构建更好的计算机的便利性，使得计算机发展到今天却具备相同的基本架构。但是也不能否认架构创新可能会成为计算机发展的新领域。

1085

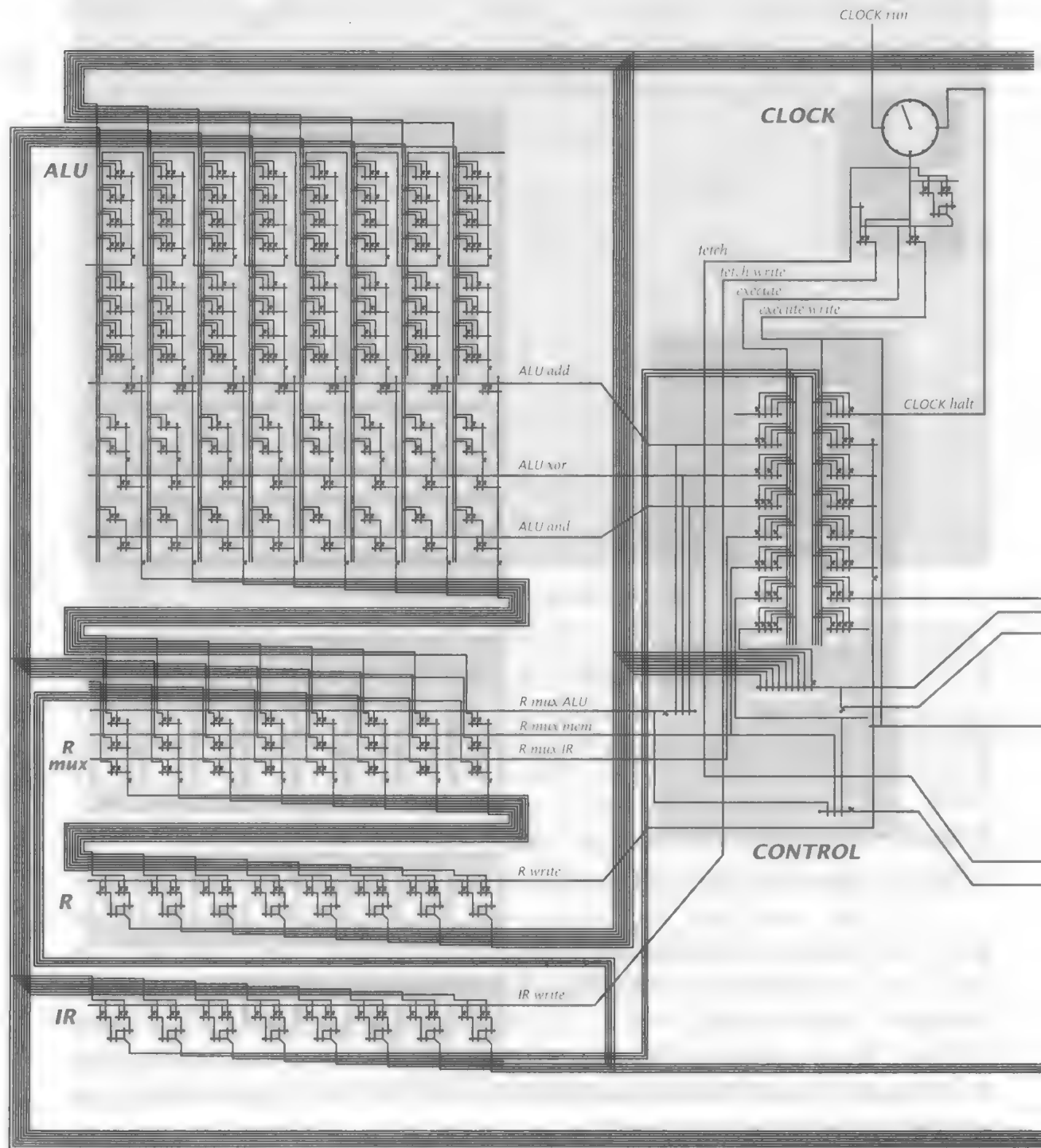
## 问答环节

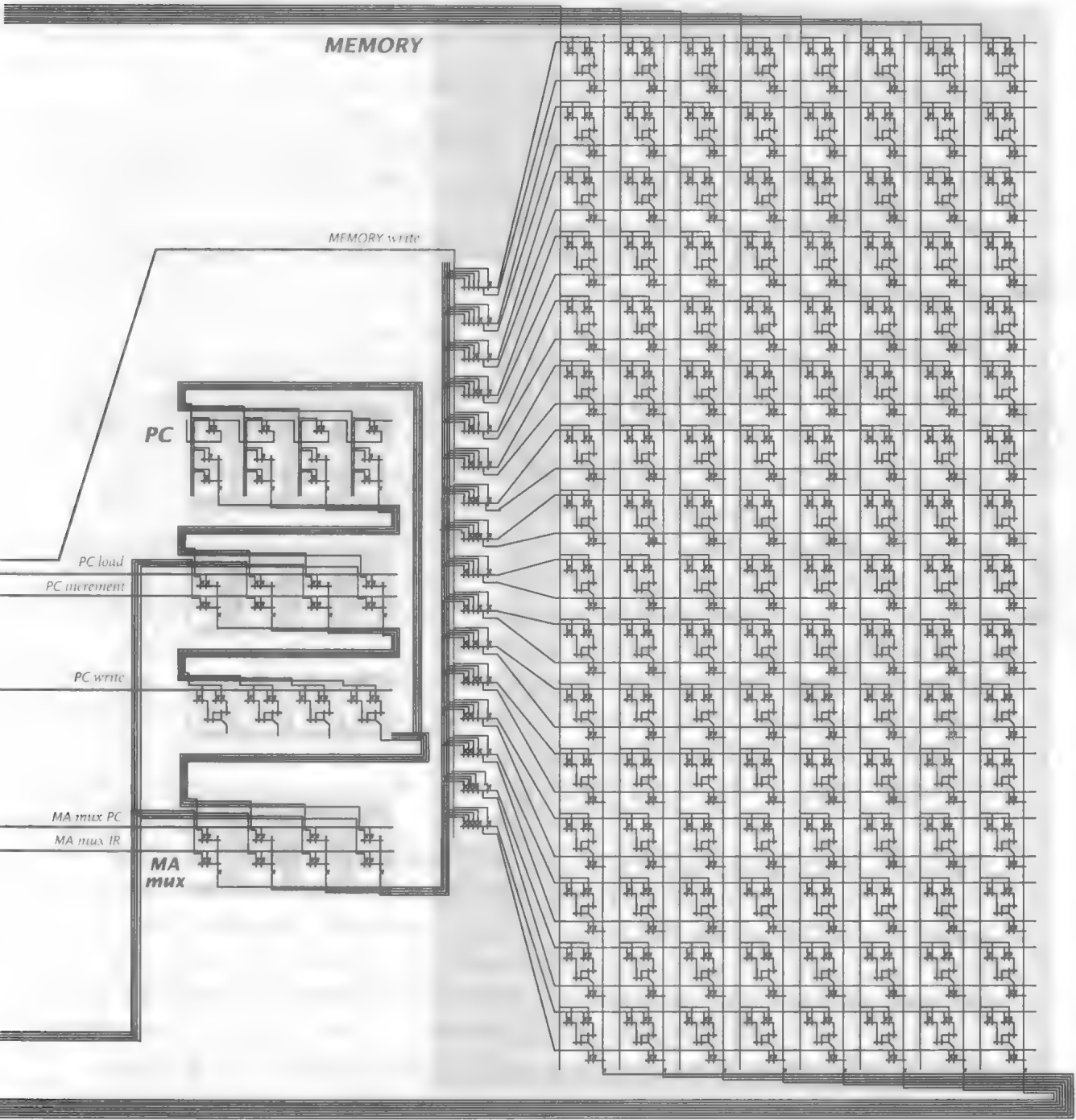
问：我不敢相信构建计算机设备如此简单。真的这就全部讲完了吗？

答：欢迎进入计算机世界！当然，在现代计算机芯片开发的过程中，需要完善许许多多细节，这也使得过去几十年中计算机得到不断的改进和深入的研究。不过，在纸上画几千条线段和点是一回事，在几平方厘米的空间内构建数十亿的物理设备，并使得它们在一秒钟内操作十亿次，则是另一回事了。

当然，我们在设计图纸上设计一个物理电路时，在实际过程中尚有许多需要改进的地方，不胜枚举。但我们讲解的模型和组件都是非常经典的，可以经受住现实和物理条件的无数次考验，并能够以多种方式实际运用。

不过，这仍然只是一个开端。自 20 世纪中叶发明第一台计算机以来，我们并未走多远。许多细节以及许多在现代 CPU 操作背后的基本设计理念依然与 TOY-8 没有什么不同。







## 习题

- 7.5.1 设计一个 TOY-8 程序，可以实现两个 8 位二进制补码数字之间的减法。
- 7.5.2 设计一个 TOY-8 程序，将小于 256 的所有斐波那契数列在标准输出上打孔显示出来。
- 7.5.3 设计一个 TOY-8 程序，可以将两个 8 位整数相乘（正负数均可）。若需要，假设内存为 32 字节。注意：为防止混淆，假设内存分成了两个区，分别是  $M0[]$  和  $M1[]$ ，这样，指令 36 就表示“将  $R$  与  $M1[6]$  的数值相加后的结果写回到  $R$  中”。
- 7.5.4 证明：内存位 0 上永远是 0。
- 7.5.5 如果在 TOY-8 机上实现 branch 和 link 指令，需要什么样的总线连接。
- 7.5.6 在我们的 TOY-8 CPU 电路里有多少个开关？设计一个表格，展示每个模块的开关数以及总数（用百分数表示）。
- 7.5.7 在本节开始，我们设计了一个计算标准输入上数字总和的 TOY-8 程序，请给出这一程序对应的控制线激活序列。

1089

## 创新练习

- 7.5.8 计数器 PC。用计数器（见练习 7.3.15）而不是增量器设计一个 4 位 PC。
- 7.5.9 设计一个计算机。参考本章结尾处的 TOY-8 的设计图，设计一个完整的计算机。可以将练习 7.3.15 和 7.3.18、7.3.8 和 7.3.11 相结合后进行扩展。

1090  
1091

# 后 记

在这里，我们先对本书中的编程和计算机科学方面的知识进行总结，然后再简单介绍今后你可能会遇到的计算机理论。若你能够通过本书体会到你身边世界中计算机理论的重要性，那将是我们莫大的荣幸。

我们在第 1 章到第 4 章中介绍了如何编程。正如你学会了如何驾驶轿车后就不会难于掌握驾驶 SUV 了，同样地，更换语言并不会导致你难以编程。许多程序员会根据不同的目的选择不同的程序语言。在第 1 章到第 3 章中，我们学习的是许多程序语言中通用的基本概念，诸如：数据类型、条件、循环、数组、第 1 章和第 2 章中的函数（在计算机发明最初的几十年中程序员一直在使用这些函数），第 3 章中的面向对象程序设计思路（这也是现代程序设计思路）。在此基础上，加上在第 4 章中学习的基本数据类型，将使你可以处理库、程序开发环境、各种专业应用等。这些也使你在设计和理解复杂系统时更深刻地体会到理论的力量。

第 5 章到第 7 章更多的是介绍了计算机科学，而非编程。在你对编程逐步熟悉和具备计算的能力后，你可以学习到 20 世纪杰出研究成果，甚而是悬而未决的难题，以及可以了解到他们在我们周围的计算设备和环境中起到的作用。正如我们在本书中不断提示的那样，计算机在对于我们理解真实世界的过程中（从基因组学到分子动力学，再到天体物理学）发挥了越来越重要的作用，而人类也必将从计算机科学中越来越多地获利。

**Java 库** Java 系统为用户提供了极为丰富的库资源，我们会大量地使用其中某些 Java 库如 Math、String 等，但也可以将其中某些忽略。Java 库的共同点之一是有大量的有关库的在线信息可供使用。若你尚未浏览 Java 库，现在就可以尝试去做。你会发现，这些代码中的大多数是供专业程序开发人员使用的，但其中也有众多库适合初学者使用。当学习库时，应保持“我可能会用到它”的心态，而非“我一定要用它”的心态。凡遇到看上去会有用的 API，就学会并掌握它！

**编程环境** 将来你肯定会遇到基于 Java 的其他编程环境。众多程序员，即便是那些富有经验的程序员，也会因为代码积累过量，而在传统的编程语言如 C 语言、C++ 语言、Fortran 语言与现代编程工具如 Ruby、Python 和 Scala 之间产生混淆。若你想学习 Python 语言，你可以参考本书的姊妹篇——《An Introduction to Programming in Python》。当你在编程时，你需要时刻记住没有哪种语言是必须使用的，若有更好的选择，那么我们就毫不犹豫地做出选择。我们坚信，固守单一的编程环境会导致你错过更好的机会。

**科学计算** 数字计算因其对准确性和精确度的要求会显得非常棘手，因此对于数学函数库的使用会特别地实用。例如，许多科学家使用的是 Fortran 语言，这是一种较为老旧的语言；还有一些科学家使用 Matlab 语言，这是一种专门为计算矩阵开发的语言。优秀的库和内置的矩阵操作结合，使得 Matlab 语言适用于众多问题。但由于 Matlab 不支持自定义数据类型和其他现代工具，使得 Java 具有更广泛的应用。而两者都可为你所用！Matlab 和 Fortran 编程人员使用的数学库与 Java（及其他现代编程语言）的库非常接近。

**应用程序 (APP) 和云计算** 如今计算机开发和使用的众多程序都可以在浏览器或移动设备上运行, 甚而是云端。而这一现象对大多数人群产生了积极的影响。若你感受到了计算这一利处, 那么很有可能你也会被本书讨论的这些方法震撼到。你可以编写程序来处理非本地的数据, 也可以编写程序在别处运行, 而这些都得力于计算设备的扩展和进化。我们的焦点放在理解这些现象的科学理论上, 这使得你可以更大规模地运用计算。

**计算机系统** 早期, 计算机系统的特点决定了它能够解决的问题的性质和外延, 但现在完全不是这样。你可以期望更大的内存、更快的设备, 尽可能保持代码机器的独立性, 努力学习和开发从 GPU 到大规模并行计算机和网络的各种新技术。

**计算理论** 与这些机会形成对比的是计算本身的各种基本限制, 这些理论的限制是计算机与生俱来的, 而且会在发展的过程中一直存在并起到关键性作用。如你所知, 我们仍然有很多问题尚未有计算机程序可以解答, 甚至也有很多问题 (如练习中列举的一些), 在可想象的计算机上也难以解决。无论是谁从事问题解决与创造性任务或研究, 若依赖于计算, 都必须承认这一事实。

**机器学习** 长久以来, 人工智能 (artificial intelligence) 都是计算机科学的一个重要领域。现代计算机的广泛应用意味着早期计算机科学家的部分梦想已经实现, 而计算机已发展到能够从其环境中实现自我学习, 从无人驾驶汽车到心仪商品推送, 甚而指导人类应该学习的内容。这一层次的学习显然要比学习任一系列的 API 或者开发任一语言都更为重要。

从尝试着编写、编译、运行第一个程序 HelloWorld 开始至今, 你已学习到了大量的知识, 但今后还有更长的路要走。一个人若坚持不懈地编程, 坚持不懈地学习编程环境、科学计算、应用程序、云计算和计算机系统、计算理论、机器学习等内容, 必然会获得难以想象的、比不懂计算机科学的人更多的机会。

# 术 语 表

**abstract machine (抽象机)**: 用于计算的数学模型。

**adder (加法器)**: 用于计算两个数字相加和的计算机组件。

**algorithm (算法)**: 用于求解某个问题的分布过程, 如欧几里得算法、归并排序和任何一种图灵机。

**alias(别名)**: 引用同一对象的一个(或多个)变量。

**alphabet (字母表)**: 一种有限的符号集, 如二进制字母 {a, b}。

**ALU (算术逻辑单元)**: 计算机的计算引擎。

**API (Application Programming Interface, 应用程序接口)**: 描述客户端如何使用一个数据类型的一系列操作的规范说明。

**array (数组)**: 用于存储一系列元素的数据结构, 支持创建、索引访问、索引赋值和迭代操作。

**argument (参数)**: 用于 Java 计算的、传递给函数的对象引用。

**ASCII (American Standard Code for Information Interchange, 美国信息交换标准代码)**: 一种广泛使用的英文文本编码标准, 被纳入 Unicode 中。

**assignment statement (赋值语句)**: 一种 Java 语句, 包括一个变量名后跟一个等号 (=), 再紧跟一个表达式, 指示 Java 对该表达式进行求值, 并把计算结果赋值给该变量。

**bit (位)**: 二进制位 (0 或 1) 之一。

**Boolean algebra (布尔代数)**: 用于布尔表达式符号操作的正式系统。

**boolean expression (布尔表达式)**: 一个表达式, 其值为 boolean 类型。

**boolean function (布尔函数)**: 将布尔值映射为布尔值的函数。

**Boolean logic (布尔逻辑)**: 布尔函数的研究

**boolean value (布尔值)**: 0 或 1, 真或假。

**booksite library (官网库)**: 本书作者创建的库, 如 StdIn、StdOut、StdDraw 和 StdAudio 等。

**built-in type (内置数据类型)**: Java 语言中内置的数据类型, 如 int、double、boolean、char 和 String。

**bus connection (总线连接)**: 具体参见 data path。

**bus mux (总线多路复用器)**: 用于切换总线连接的多路复用器。

**buzzer(蜂鸣器)**: 电路中的一个不稳定的反馈回路。

**Church-Turing thesis (邱奇-图灵论题)**: 任何在物理真实计算设备上可以描述的问题可以由图灵机计算 (理解一门语言或计算一个函数)。

**circuit (电路)**: 导线、电源连接和开关的互连网络。

**class(类)**: 用于实现自定义数据类型的 Java 构造, 可以提供模板来创建和操作包含该类型值的对象, 由 API 定义其实现的程序。

**.class file (.class 文件)**: 扩展名为 .class 的文件, 包含 Java 字节码, 适合在 Java 虚拟机上执行。

**class variable (类变量)**: 具体参见 static variable。

**client (客户端)**: 通过 API 使用其实现的程序。

**clock (时钟)**: 一种时序电路, 用于对处理器中的取指和执行控制线进行排序。

**combinational circuit (组合电路)**: 没有环路的电路。

**command line (命令行)**: 终端应用程序的当前活动行, 用于调用系统命令的运行程序。

**command-line argument (命令行参数)**: 在命令行传递给程序的字符串。

**comment (注释)**: 用于帮助读者理解代码目的的解释性文本 (被编译器忽略)。

**comparable data type (可比较的数据类型)**: 一种 Java 数据类型, 它实现了 Comparable 接口

并定义了全序关系。

**compile-time error** (编译错误): 由编译器发现的错误。

**compiler** (编译器): 把一个程序从高级语言翻译成低级语言的程序。Java 编译器将 .java 文件 (包含 Java 源代码) 翻译为 .class 文件 (包含 Java 字节码)。

**computability** (可计算性): 在计算机上解决问题的能力, 某些问题 (如停机问题) 是不可计算的。

**conditional statement** (条件语句): 一种根据一个或多个布尔表达式的值来执行不同计算的语句, 如 if、if-else 或 switch 语句。

**constant variable** (常量): 在程序编译时已知且在程序执行过程中 (或从程序的一次执行到下一次执行期间) 保持不变的变量。

**constructor** (构造函数): 一种用于创建和初始化一个新对象的特殊方法。

**control** (控制): 组合电路, 用于组织处理器中的控制线。

**control line** (控制线): 带有控制信号 (与数据值相对) 的导线。

**controlled switch** (控制开关): 可以断开连接的电路元件。

**CPU** (Central Processing Unit): 实现计算机的电路。

**data path** (数据路径): 连接一个模块到另一个模块的一组电线 (也称为总线连接)。

**data structure** (数据结构): 在计算机中组织数据的方式 (通常用于节省时间或空间), 如数组、可变数组、链表和二叉搜索树。

**data type** (数据类型): 一系列值的集合以及定义在这些值上的一系列操作的集合。

**declaring a variable** (声明一个变量): 指定一个变量的名称和类型。

**decoder** (解码器): 用于选择输出的组合电路。

**demultiplexer** (多路分配器): 一种组合电路, 用于将输入切换到所选择的输出。

**deterministic** (确定性): 一种计算, 其中每个步骤完全由当前状态决定。

**deterministic finite-state automaton** (确定有限状态自动机, DFA): 能够识别常规语言的确定性抽象机器。

**element** (元素): 数组中的一个对象。

**evaluate an expression** (表达式求值): 通过在表达式中应用运算符计算操作数, 把一个表达式简化为一个值。运算符优先级、运算符结合律和计算顺序决定了将运算符应用于操作数的顺序。

**exception** (异常): 程序运行时的一种异常情况或错误。

**exponential-time algorithm** (指数运行时间算法): 一种以输入规模的指数函数为运行时间限制的算法。

**expression** (表达式): 字面量、变量、运算符和函数调用的组合, Java 可以将其化简为一个值。

**fetch-increment-execute cycle** (取指-递增-执行周期): 处理计算机操作的过程。

**flip-flop** (触发器): 一种时序电路, 可以实现内存位。

**floating point** (浮点): 使用“科学记数法”来表示计算机上实数的一般描述 (具体参见 IEEE 754)。

**formal language** (形式语言): 给定字母表上的一系列字符串。

**function** (函数): 具体参见 static method。

**functional interface** (函数式接口): 只包含一个抽象方法的接口。

**garbage collection** (垃圾回收): 自动识别不再使用的对象并释放其占用内存的过程。

**gate** (门): 一种小型组合电路, 用于实现一个布尔函数, 如与门、或门、非门等。

**generalized regular expression pattern match** (广义正则表达式模式匹配, grep): 一个使用正则表达式搜索模式的经典方法。

**generic class** (泛型类): 由一个或多个类型参数参数化的类, 如 Queue、Stack、ST 或 SET。

**global variable** (全局变量): 一个作用范围是整个程序和文件的变量, 具体参见 static variable。

**halting problem (停机问题):** 图灵机不可解问题。

**hashing (散列):** 将数据类型值转换为给定范围内的整数, 使得不同的键不太可能映射到同一整数。

**hash table (散列表):** 基于散列的符号表实现。

**hexadecimal (十六进制):** 整数的 Base-16 表示。

**identifier (标识符):** 用于识别变量、函数、类、模块和其他对象的名称。

**IEEE 754:** 浮点计算的国际标准, 用于现代计算机硬件 (具体参见 floating point)。

**immutable data type (不可变数据类型):** 任何实例的值不可改变的数据类型, 如 Integer、String 或 Complex。

**immutable object (不可变对象):** 值不可改变的对象。

**implementation (实现):** 实现由 API 定义的一系列方法的程序, 可被客户端使用。

**import statement (导入语句):** 可引用其他模块中代码的一种 Java 语句, 可以不使用其完全限定名。

**initializing a variable (初始化变量):** 在程序第一次给变量赋值。

**instance (实例):** 特定类的一个对象。

**instance method (实例方法):** 数据类型操作的实现 (针对特定对象调用的函数)。

**instance variable (实例变量):** 在类中定义的变量 (但在方法之外), 表示数据类型的值 (与类的每个实例关联)。

**instruction register (指令寄存器, IR):** 保存正在执行的指令的机器组件。

**interface (接口):** 类的协议, 用于实现一系列方法。

**interpreter (解释器):** 逐行执行使用高级语言编写的程序的一种程序, Java 虚拟机解释 Java 字节码并在你的计算机上执行它。

**intractability (难解性):** 在多项式时间内解决计算机上的问题的不可能程度。

**item (项):** 集合中的一个对象。

**iterable data type (可迭代数据类型):** 一种实现 Iterable 接口的数据类型, 可以与 foreach 循环一起使用, 如 Stack、Queue 或 SET。

**iterator (迭代器):** 实现 Iterator 接口的数据类型。用于实现可迭代的数据类型。

**Java bytecode (Java 字节码):** Java 虚拟机使用的低级、与机器无关的语言。

**.java file (.java 文件):** 一个文件, 包含用 Java 程序设计语言编写的程序。

**Java programming language (Java 程序设计语言):** 一种通用的、面向对象的程序设计语言。

**Java Virtual Machine (Java 虚拟机, JVM):** 在微处理器上执行 Java 字节码的程序, 同时使用解释器和即时编译器。

**just-in-time-compiler (即时编译器):** 在程序执行时, 编译器将高级语言的程序连续转换为低级语言。Java 的即时编译器从 Java 字节码转换为本机机器语言。

**Kleene's theorem (克林定理):** 正则表达式 (RE)、确定有限状态自动机 (DFA) 和非确定有限状态自动机 (NFA) 都具有正则语言的特征的概念。

**lambda calculus (Lambda 演算):** 阿隆索引入的通用计算模型。

**lambda expression (Lambda 表达式):** 一个匿名函数, 可以在以后传递和执行。

**library (库):** 一个 .java 文件, 具有允许其功能在其他 Java 程序中复用的结构。

**linked list (链表):** 由一系列节点组成的数据结构, 其中每个节点包含对序列中下一个节点的引用。

**literal (常量):** 用于内置数据类型值的源代码表示方式, 如 123、'Hello' 和 True。

**local variable (局部变量):** 在一个方法中定义的变量, 其作用范围仅限于该方法。

**loop (循环):** 一种语句, 它根据某些布尔表达式的值执行重复计算, 如 for 或 while 语句。

**machine-language instruction (机器语言指令):** 内置于计算机硬件中的操作, 如 TOY 中的添加、加载和分支 0。

**machine-language program (机器语言程序):** 在计算机上执行的一系列机器语言指令。

**masking (屏蔽):** 屏蔽计算机一字节中的一些位。

- memory (内存)**: 计算机的组成部分, 用于加载数据和指令。
- method (方法)**: 一个被命名的编程语句序列, 可以被其他代码调用以执行计算。
- method call (方法调用)**: 一个表达式, 用于执行方法并返回。
- modular programming (模块化程序设计)**: 一种编程风格, 强调使用分离、独立的模块解决任务。
- module (模块, 软件层面)**: 实现一个 API 的独立程序, 如 Java 类。
- module (模块, 硬件层面)**: 计算机组件, 通常具有总线输入和输出。
- Moore's law (摩尔定律)**: 戈登·摩尔 (Gordon Moore) 观察到这一现象, 自 20 世纪 60 年代引入集成电路以来, 处理器功率和存储器容量每两年翻一番。
- multiplexer (多路复用器)**: 一种组合电路, 可将选定的输入切换为输出。
- mutable data type (可变数据类型)**: 一种数据类型, 其实例的值可以更改, 如 Counter、Picture 或数组。
- mutable object (可变对象)**: 数据类型值可以更改的对象。
- nondeterministic (不确定性)**: 可以从一个或多个状态进行多个后续步骤的计算。
- nondeterministic finite-state automaton (非确定有限状态自动机, NFA)**: 一种识别形式语言的非确定性抽象机器。
- NP (NP 类)**: 一个集合, 包括所有搜索问题, 如布尔方程是否可解、因式分解和 P 集合中的所有问题。
- NP-complete (NP 完全问题)**: NP 中“最难”问题的代表, 如布尔方程可解性、顶点覆盖问题和整数线性不等式可解性。
- null reference (空引用)**: 特殊的字面量 null, 表示对空对象的引用。
- object (对象)**: 特定数据类型的一个值在计算机内存中的表示, 其特性包括状态 (数据类型值)、行为 (数据类型操作) 和标识 (存储器中的位置)。
- object-oriented programming (面向对象程序设计)**: 一种编程风格, 强调使用数据类型和对象来对现实世界或抽象实体进行建模。
- object reference (对象引用)**: 对象标识的具体表示 (对象存储的内存地址)。
- operand (操作数)**: 运算符作用的对象。
- operating system (操作系统)**: 运行在计算机上的程序, 用于管理资源, 为各种程序和应用提供通用服务。
- operator (运算符)**: 表示内置数据类型操作的一种特殊的符号 (或一系列符号), 如 +、-、\* 和 []。
- operator associativity (运算符结合律)**: 一种规则, 用于确定在使用具有相同优先级的运算符时的顺序, 如 1-2-3。
- operator precedence (运算符优先级)**: 一种规则, 用于确定在表达式中应用运算符的顺序, 如  $1 + 2 * 3$ 。
- order of evaluation (计算顺序)**: 子表达式中的计算顺序。无论运算符优先级还是运算符结合律, Java 都会从左到右计算子表达式。在调用方法之前, Java 会从左到右计算方法参数。
- overflow (溢出)**: 当算术运算结果的值超过最大可能值时会导致溢出。
- overloading a method (重载一个方法)**: 使用相同名称 (但不同的参数列表) 定义两个或多个方法。
- overloading an operator (重载一个运算符)**: 为一个数据类型定义一个运算符 (如 +、\*、<= 和 []) 的行为, Java 不支持运算符重载。
- overriding a method (重写一个方法)**: 重新定义继承的方法, 如 equals() 和 hashCode()。
- P (P 集合)**: 一个集合, 包括多项式时间算法的所有搜索问题, 如排序、最短路径和线性不等式可解性。
- P≠NP conjecture (P≠NP 猜想)**: 一个众所周知的猜想是: 一些搜索问题无法在多项式时间内被解决。
- package (包)**: 共享公共命名空间的相关类和接



- 口的集合。包 `java.lang` 包含最基本的类和接口，并自动导入；包 `java.util` 包含 Java 的 Collections Framework。
- paper tape (纸带)**：一种早期原始的输入 / 输出媒介。
- parameter variable (参数变量)**：在函数定义中指定的变量，调用函数时初始化为对应的参数。
- parsing (分析)**：将字符串转换为内部表示。
- pass by value (按值传递)**：Java 将方法传递给方法的方式，即通过作为数据类型值（对于基本类型）或作为对象引用（对于引用类型）的方式传递。
- PDP-8**：20 世纪 70 年代使用的真正计算机。
- polymorphism (多态性)**：不同的数据类型使用相同的 API（或部分 API）。
- polynomial-time algorithm (多项式时间算法)**：能够保证在输入规模的某些多项式函数时间内运行的算法。
- polynomial-time reduction (多项式时间归约)**：使用子程序解决一个问题，以帮助有效地解决另一个问题。
- primitive data type (基本数据类型)**：Java 定义的八种数据类型之一，包括 `boolean`、`char`、`double` 和 `int`。基本类型的变量存储数据类型值本身。
- private (私有)**：不被客户端引用的数据类型的实现代码。
- program (程序)**：在计算机中执行的指令序列。
- program counter (程序计数器, PC)**：保存下一条要执行的指令的地址的机器组件。
- pure function (纯函数)**：给定相同的参数，时钟返回相同值且不产生任何可观察到的副作用的函数。
- reduction (归约)**：使用子程序解决另一个问题。
- reference type (引用类型)**：一种类型、接口类型或数组类型，如 `String`、`Charge`、`Comparable` 和 `int []`。引用类型的变量存储的是对象引用，而不是数据类型值本身。
- register (寄存器)**：包含要处理的内容的机器组件。
- regular expression (正则表达式, RE)**：一个表达式，它使用并集、连接、闭包和括号来规定形式语言。
- regular language (正则语言)**：可由 DFA 识别或由正则表达式规定的语言。
- resizing array (可变数组)**：一种数据结构，用于确保使用数组的恒定比例部分被使用。
- return value (返回值)**：作为函数调用结果提供给调用者的值。
- run-time error (运行时错误)**：程序执行时发生的错误。
- satisfiability (可满足性)**：对于一组给定的方程（或不等式），是否存在一组变量的值，使得这组方程（或不等式）为真。
- scope of a variable (变量的作用域)**：一个变量或名称可以被直接访问的程序区域。
- search problem (搜索问题)**：一个问题，存在多项式时间的算法，可以检查任何声称有效的解决方案是否确实有效。
- side effect (副作用)**：一种状态的改变，如打印输出、读取输入、抛出错误或更改某些持久对象（实例变量、参数变量或全局变量）的值。
- sequential circuit (时序电路)**：具有环路（反馈）的电路。
- source code (源代码)**：高级程序设计语言（如 Java）中的程序或程序片段。
- standard input, output, drawing, and audio (标准输入、标准输出、标准绘图、标准音频)**：本书提供的 Java 输入输出模块。
- statement (语句)**：Java 可以执行的一条指令，如赋值语句、`if` 语句、`while` 语句和 `return` 语句。
- static method (静态方法)**：在 Java 类中实现的函数，如 `Math.abs()`、`Euclid.gcd()`、`StdIn.readInt()`。
- static variable (静态变量)**：与类关联的变量。
- string (字符串)**：有限的字母符号序列。
- sum-of-products representation (“积之和”表示)**：布尔函数的标准代数表示形式。
- terminal window (终端窗口)**：接受命令的操作系统的应用程序。
- this**：在实例方法或构造函数中，引用正在调用其方法或构造函数的对象的关键字。

**throw an exception** (抛出一个异常): 发出编译时或运行时错误信号。

**TOY**: 一个虚构的计算机, 类似于 PDP-8, 专为本书而设计。

**trace** (追踪): 逐步描述程序的操作。

**Turing machine** (图灵机, TM): 由艾伦·图灵引入的通用计算模型。

**two's complement representation** (二进制补码表示): 在计算机中表示负整数的公约。

**type parameter** (类型参数): 泛型类中的占位符, 用于客户端指定的某种具体类型。

**Unicode**: 文本编码的国际标准。

**unit testing** (单元测试): 在每个模块中包含用于测试其代码的实践方法。

**universal Turing machine** (通用图灵机, UTM): 一种图灵机, 可以在任意输入上模拟任意图灵机。

**universality** (普遍性): 所有足够强大的计算设备可以决定同一组形式语言并计算同一组数学函数的想法。

**variable** (变量): 拥有值的实体。每个 Java 变量都包含名称、类型和范围。

**virtual machine** (虚拟机): 作为另一台计算机上程序的计算机的定义或实现。

**von Neumann machine** (冯·诺依曼机器): 计算机的体系结构, 其中指令和数据存储在单一存储器中。

**wire** (导线): 带有布尔值的电路元件。

**word** (字): 固定长度的位序列, 作为计算机体系结构中的一个单元处理。

**wrapper type** (封装类型): 与其中一种基本类型对应的引用类型, 如 Integer、Double、Boolean 和 Character。

# 索引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

## A

- A-format instructions (A 格式指令), 911
- Absolute value function (绝对值函数), 199
- Absorption identity (吸收律恒等式), 990
- Abstract machines (抽象机), 737–738
- Abstract methods (抽象模型), 446
- Abstraction (抽象)
  - color (颜色), 341–343
  - circuits (电路), 1037–1039
  - data (数据), 382
  - displays (显示), 346
  - function-call (函数调用), 590–591
  - libraries (库), 230, 429
  - object-oriented programming (面向对象编程), 329
  - printing as (打印), 76
  - recursion (递归), 289
  - vs. representation (各种表示的对比), 69
  - standard audio (标准音频), 155
  - standard drawing (标准绘图), 144
  - standard I/O (标准 I/O), 129, 139–143
- Accept states (接受状态)
  - DFA (确定有限状态自动机), 738–739
  - Turing machines (图灵机), 766–767
- Access modifiers (访问修饰符), 384
- Accessing references (对象访问), 339
- Account information (账户信息)
  - dictionary lookup (字典查找), 628–629
  - indexing (索引), 634
- Accuracy (精度)
  - n-body simulation (多体模拟), 488
  - random web surfer (随机网络冲浪), 185
- Adaptive plots (自适应绘图), 314–318
- Adders (加法器)
  - binary (二进制), 771
  - combinational circuits (组合电路), 1007
  - overview (概述), 1028
  - ripple-carry (进位加法器), 1028–1030
  - sum-of-products (积之和), 1028
- AddInts program (程序 AddInts), 134
- Addition (加法)
  - complex numbers (复数), 402–403
  - floating-point numbers (浮点数), 24–26
  - integers (整数), 22, 884
  - negative numbers (负数), 887
  - spatial vectors (空间向量), 442–443
- Address control lines (地址控制线), 1056
- Addresses (地址)
  - array elements (数组元素), 94
  - memory (内存), 909
  - symbolic names (符号名称), 981
- Adelman, Leonard (伦纳德·阿德尔曼), 795
- Adjacency matrix (邻接矩阵), 692
- Adjacent vertices (邻接顶点), 671
- Albers, Josef (约瑟夫·亚伯斯), 342
- AlbersSquaresprogram (程序 AlbersSquares), 341–342
- Alex (亚历克斯), 380
- Algebra (代数)
  - boolean (布尔), 989–991
  - vectors (向量), 442–443
- Algorithms (算法), 493
  - computability (可计算性), 787
  - decidability (可判定性), 786–787
  - exponential-time (指数时间), 826
  - overview (概述), 786
  - performance (性能)
  - polynomial-time (多项式时间), 825–826
  - searching (搜索)
  - sorting (排序)
- Aliasing (别名)
  - arrays (数组), 516
  - bugs from (错误), 439, 441
  - references (引用), 363

- Allocating memory (分配内存), 94, 367
- Alphabets (字母表)
  - formal languages (形式语言), 720–721
  - metasymbols (元符号), 725
  - regular expressions (正则表达式), 730
  - symbols (符号表), 718–719
- ALUs (算术逻辑单元)
- Amortized analysis (摊销分析), 580–581
- Ampersands (&)
  - bitwise operations (按位运算), 891–892
  - boolean type (布尔类型), 26–27, 991
- Analog circuits (模拟电路), 1013
- AND circuits in ALUs (ALU 中的与电路), 1031
- AND gates (与门), 1014
- And operation (与运算符)
  - bitwise (按位), 891–892
  - boolean type (布尔类型), 26–27, 987–989
  - TOY machine (TOY 机), 913
- Animations (动画)
  - BouncingBall, 152–153
  - double buffering (双缓冲区), 151
- Annihilation identity (0-1 律定义), 990
- Antisymmetric property (反对称属性), 546
- Application programming interfaces (应用程序编程接口, API)
  - access modifiers (访问修饰符), 384
  - Body, 480
  - built-in data types (内置数据类型), 30–32
  - Charge, 383
  - Color, 343
  - Comparable, 545
  - Complex, 403
  - Counter, 436–437
  - data types (数据类型), 388
  - designing (设计), 233, 429–431
  - Draw, 361
  - Graph, 675–679
  - Histogram, 392
  - implementing (实施), 231
  - In, 354
  - libraries (库), 29, 230–232
  - modular programming (模块化编程), 432
  - Out, 355
  - PathFinder, 683
  - Picture, 347
  - Queue, 592
  - SET, 652
  - Sketch, 459
  - spatial vectors (空间向量), 442–443
  - ST, 625
  - StackOfStrings, 568
  - StdArray, 237
  - StdAudio, 159
  - StdDraw, 149, 154
  - StdIn, 132–133
  - StdOut, 130
  - StdRandom, 233
  - StdStats, 244
  - StockAccount, 410
  - Stopwatch, 390
  - String, 332–333
  - symbol tables (符号表), 625–627
  - Turtle, 394
  - Universe, 483
  - Vector, 443
- Approximation algorithms (近似算法), 852
- Arbitrary-size input streams (任意大小的输入流), 137–138
- args argument (args 参数), 7, 208
- Arguments (实参)
  - arrays as (数组), 207–210
  - command-line (命令行), 7–8, 11, 127
  - constructors (构造函数), 333, 385
  - methods (方法), 30
  - passing (传递), 207–210, 364–365
  - printf(), 130–132
  - static methods (静态方法), 197
- Ariane 5 rocket (Ariane 5 运载火箭), 35
- Arithmetic (算术)
  - CPU instructions (CPU 指令), 1079
  - floating point numbers (浮点数), 890
  - integers (整数), 884–885
  - operators (运算符), 22
  - TOY machine instructions (TOY 机器指令), 912
- Arithmetic logic units (算术逻辑单元, ALUs), 1031
  - bitwise operations (按位运算), 1031
  - inputs (输入), 1031
  - outputs (输出), 1032

- summary (总结), 1032–1033
- TOY machine (TOY 机), 910
- Arithmetic expression evaluation (算术表达式计算), 586–589
- Arithmetic shifts (算术移位)
  - bits (位), 891–892
  - purpose (目的), 898–899
- ArrayIndexOutOfBoundsException, 95, 116, 466
- Arrays (数组)
  - aliasing (别名), 516
  - as arguments (作为参数), 207–210
  - assigning (赋值), 117
  - associative (关联), 630
  - binary searches (二进制搜索), 538–539
  - bitonic (双调), 563
  - bounds checking (边界检查), 95
  - comparing (比较), 117
  - coupon collector problem (卡片收集问题), 101–103
  - decks of cards (卡片组), 97–100
  - declaring (声明), 91, 116
  - default initialization (默认初始化), 93
  - exchanging values (交换值), 96
  - FIFO queues (FIFO 队列), 596
  - hash tables (散列表), 636
  - I/O libraries (I/O 库), 237–238
  - images (图像), 346–347
  - immutable types (不可变类型), 439–440
  - iterable classes (可迭代的类), 603
  - linked structures (链接结构), 942–944
  - machine-language (机器语言), 938–941
    - memory (内存), 91, 94, 515–517
  - multidimensional (多维), 111
  - overview (概述), 90–92
  - parallel (平行), 411
  - plotting (绘制), 246–248
  - precomputed values (预先计算的值), 99–100
  - references (引用), 365
  - resizing (可变), 578–581, 635
    - as return values (作为返回值), 210
  - setting values (设定值), 95–96
  - shuffling (混排), 97
  - side effects (副作用), 208–210
  - Sieve of Eratosthenes (埃拉托斯特尼筛法), 103–105
  - stacks (栈), 568–570, 578–581
  - summary (总结), 115
  - transposition (交换), 120
  - two-dimensional (二维的)
- Arrays.binarySearch(), 559
- Arrays.sort(), 559
- ArrayStackOfStrings program (程序 ArrayStackOfStrings), 568–570, 603
- Arrival rate in M/M/1 queues (M/M/1 队列的到达率), 597–598
- The Art of Computer Programmingbook (计算机编程艺术), 947
  - ASCII standard (ASCII 标准), 874, 894–895
  - Assemblers for TOY machine (TOY 机的汇编), 964
  - Assembly language (汇编语言)
    - description (描述), 930
    - symbolic names (符号名称), 981
- Assertions (断言), 466–467
- Assignments (赋值)
  - arrays (数组), 117
  - chained (链式), 43
  - compound (复合), 60
  - description (描述), 17
  - references (引用), 363
- Associative arrays (关联数组), 630
- Associative axiom (结合律公理), 990, 993
- Associativity (结合律), 17
- Asterisks (星号, \*)
  - comments (注释), 9
  - floating-point numbers (浮点数), 24–26
  - integers (整数), 22–23
  - regular expressions (正则表达式), 724
- Audio (音频)
  - plotting sound waves (绘制音频波形), 249
  - standard (标准), 155–159
  - superposition (叠加), 211–215
- Autoboxing (自动装箱), 457, 585–586
- Automatic promotion (自动提升), 33
- Average-case performance (平均情况表现), 648
- Average magnitude (平均幅度), 164
- Average path lengths (平均路径长度), 693
- Average power (平均功率), 164
- Average program (程序 Average), 137–138
- Axioms in Boolean algebra (布尔代数中的公理), 990

## B

- Backslashes (反斜杠, \)
  - escape sequences (转义序列), 19
  - regular expressions (正则表达式), 731
- Backward compatibility (向后兼容性), 976
- Bacon, Kevin (凯文·贝肯), 684
- Balanced binary trees (平衡二叉树), 661
- Ball animation (小球动画), 152–153
- Barnsley ferns (巴恩斯利蕨), 240–243
- Base cases (基础步骤)
  - binary search trees (二分查找法), 640
  - recursion (递归), 264–265, 281
- Base classes (基类), 452–453
- Base64 encoding (Base64 编码), 904
- Bases in positional notation (按位计数法的基数), 875
- Basic scaffolding (基础脚手架), 302–304
- Basic statistics (基础统计), 244–246
- Beck exploit (beck 漏洞攻击), 529
- Beckett, Samuel (塞缪尔·贝克特), 273
- Beckett program (Beckett 程序), 274–275
- Behavior of objects (对象的行为), 340
- Benford's law (本福德定律), 224
- Bernoulli, Jacob (雅各布·伯努利), 398
- Bernoulli program (Bernouli 程序), 249–250
- Best-case performance (最佳表现)
  - binary search trees (二叉搜索树), 647
  - insertion sort (插入排序), 544
- Big-O notation (大 O 表示法), 520–521
- BigInteger class (BigInteger 类), 827, 897–898
- Binary adders (二进制加法器), 771
- Binary digits (二进制数字), 22
- Binary frequency count equality (二进制频率计数相等), 772–773
- Binary incrementers (二进制增量器), 769–771
- Binary number system (二进制系统)
  - conversions (转换), 67–69
  - description (描述), 38
- Binary operators (二元操作符), 17
- Binary program (Binary 程序), 67–69
- Binary reflected Gray code (二进制表示的格雷码), 274
- Binary representation (二进制表示)
  - decimal conversions (十进制转换), 877
  - description (描述), 875
  - examples (示例), 878–879
  - hex conversions (十六进制转换), 876–877
  - literals (常量), 891
- Binary search trees (二叉搜索树, BSTs)
  - implementation (实现), 645–646
  - insert process (插入过程), 644–645
  - machine-language (机器语言), 942–944
  - ordered operations (与顺序相关的操作), 651
  - overview (概述), 640–643
  - performance (性能), 647–648
  - search process (搜索过程), 643–644
  - symbol tables (符号表), 624–625
  - traversing (遍历), 649–650
- Binary searches (二分查找)
  - binary representation (二进制表示), 536
  - correctness proof (算法正确性证明), 535
  - exception filters (异常过滤器), 540
  - inverting functions (求反函数), 536–538
  - overview (概述), 533–534
  - random web surfer (随机网上冲浪), 176
  - running time (运行时间), 535
  - sorted arrays (排序数组), 538–539
  - symbol tables (符号表), 635
  - weighing objects (物体称重法), 540–541
- Binary strings (二进制字符串), 718–719
- Binary trees (二叉树)
  - balanced (平衡), 661
  - heap-ordered (堆排序), 661
  - isomorphic (同构), 661
- Binary16 format (Binary 16 格式), 888
- BinarySearch program (程序 BinarySearch), 538–539
- Binomial coefficients (二项式系数), 125
- Binomial distributions (二项式分布), 125, 249
- Biology (生物学)
  - computational (计算), 732–734
  - DNA computers (DNA 计算机), 795
  - genomics application (基因组学应用), 336–340
  - graphs (图谱), 672
- Bipartite graphs (二分图), 682
- Bisection searches (二分搜索), 537
- Bit-slice memory design (位片内存设计), 1054–1056

- Bitmapmed images (位图图像), 346
- Bitonic arrays (双调数组), 563
- Bits (位)
  - binary number system (二进制系统), 38, 875
  - bitwise operations (按位运算), 891–892
  - computer dependence (计算机依赖), 874
  - description (描述), 22
  - logical instructions (逻辑指令), 912–913
  - manipulating (操作), 891–893
  - memory (内存), 1056
  - memory size (内存大小), 513
  - register (寄存器), 1051
  - shifting (移位), 891–892
- Bitwise operations (按位运算)
  - and (与), 913
  - arithmetic logic units (算术逻辑单元), 1031
  - exclusive or (异或), 39, 913
  - shift (移位), 913
- Black–Scholes formula (布莱克–斯科尔斯期权估价公式), 222, 565
- Blobs, 709
- Blocks (块)
  - statements (语句), 50
  - variable scope (变量作用域), 200
- Bodies (体)
  - loops (循环), 53
  - static methods (静态方法), 196
- Body program (程序 Body)
  - memory (内存), 514
  - N-body simulation (多体模拟), 479–482
- Bolloba s–Chung graph model (Bolloba s–Chung 图模型), 713
- Book indexes (本书索引), 632–633
- Booksite (本书官网), 2–3
- Boole, George (布尔·乔治), 986
- boolean data type (布尔数据类型)
  - conversion codes (转换代码), 131–132
  - description (描述), 14–15
  - input (输入), 133
  - memory size (内存大小), 513
  - overview (概述), 26–27
- Boolean logic (布尔逻辑)
  - cryptography application (密码学应用), 992–994
  - description (描述), 27
  - expressions (表达式), 995–996
  - functions (函数), 987–991, 994–997
  - overview (概述), 986
- Boolean matrices (布尔矩阵), 302
- Boolean satisfiability (布尔可满足性), 832, 836
  - boolean equation satisfiability problem (布尔方程可满足性问题), 838
  - NP-completeness (NP 完全性), 844–846, 853–856
- Booting (启动), 959–960, 968–969
- Bootstrapping (步步为营法), 971
- BouncingBall program (程序 BouncingBall), 152–153
- Bounding boxes for drawings (绘图边界框), 146
- Bounds (边界)
  - arrays (数组), 95
  - exponential time (指数时间), 826
  - polynomial time (多项式时间), 825
- Boxing (装箱), 457, 585–586
- Box–Muller formula (Box–Muller 公式), 47
- Breadth-first searches (广度优先算法), 683, 687–688, 690, 692
- break statement (break 语句), 74
- Bridges, Brownian (布朗桥), 278–280
- Brin, Sergey (谢尔盖·布林), 184
- Brown, Robert (罗伯特·布朗), 400
- Brownian bridges (布朗桥), 278–280
- Brownian motion (布朗运动), 400–401
- Brownian program (Brownian 程序), 278–280
- Brute-force algorithm (暴力算法), 535–536
- BST program (程序 BST), 645–646
- BSTs
- Buffer overflow (缓冲区溢出)
  - arrays (数组), 95
  - attacks (攻击), 963
- Buffering drawings (绘图缓冲区), 151
- Bugs (错误)
  - aliasing (别名), 363, 439, 441
  - overview (概述), 6
  - testing for (测试), 318
- Built-in data types (内置数据类型)
  - boolean (布尔数据类型), 26–27
  - characters and strings (字符和字符串), 19–21



comparisons (比较), 27–29  
 conversions (转换), 32–35  
 floating-point numbers (浮点数), 24–26  
 integers (整数), 22–24  
 library methods (库方法), 29–32 overview, 14–15  
 summary (总结), 35–36  
 terminology (术语), 15–18  
 Built-in interfaces (内置接口), 451  
 Buses (总线), 1034–1036  
   CPU connections (CPU 连接), 1077  
   program counter connections (程序计数器连接), 1073–1074  
 Buzzers (蜂鸣器), 1048  
   byte data type (字节数据类型), 24  
 Bytecode (字节码)  
   compiling (编译), 589, 788  
   Java virtual machine (Java 虚拟机), 965  
 Bytes memory size (字节内存大小), 513

## C

C conversion specification (C 转化规范), 131  
 Caches (缓存)  
   and instruction time (指令时间), 509  
   in top-down dynamic programming (自上而下动态编程), 284  
 Calculators (计算器), 908  
   Callbacks in event-based programming (基于事件编程的回调), 451  
 Calls (调用), 193  
   chaining (链接), 404  
   in machine language (使用机器语言), 932–933  
   methods (方法), 30, 197, 340  
   reverse Polish notation (反向波兰表示法), 591  
 Canvas (画布), 151  
 Card decks, arrays for (扑克牌, 数组), 97–100  
 Carets (脱字符运算 ^)  
   bitwise operations (按位运算), 891–892  
   regular expressions (正则表达式), 731  
 Carroll, Lewis (刘易斯·卡罗尔), 710  
   Carry bits in adders (加法器中进位), 1028  
   Cartesian representation (笛卡儿表示), 433  
 Casts (转型), 33–34  
 Cat program (cat 程序), 356  
 Cellular automata (元胞自动机), 794

Central processing units (中央处理器, CPUs)  
   bus connections (总线连接), 1077  
   control lines (控制线), 1077–1078  
   execute phase (执行阶段), 1079  
   fetch phase (取指阶段), 1078  
   instructions (说明), 1079–1080  
   interfaces (接口), 1076  
   load address (加载地址), 1080  
   modules (模块), 1076  
   overview (概述), 985  
   TOY-8 machine (TOY-8 机), 1076–1080  
 Centroids (质心), 164  
 Chained assignments (链式赋值), 43  
 Chained comparisons (链式比较), 43  
 Chaining method calls (链式方法调用), 404  
 Characters and char data type (字符与字符串类型)  
   ASCII (ASCII 编码), 894  
   conversion to numbers (转换为数字), 880–881  
   description (描述), 15  
   memory size (内存大小), 513  
   representing (表示), 894–895  
   Unicode, 894–895  
   working with (协作), 19–21  
 Charge program (程序 Charge), 383–389, 515  
 Checksums (校验和)  
   description (描述), 86  
   formula (公式), 220  
 Chords (和弦), 211  
 Chromatic scale (半音阶), 156  
 Church, Alonso (阿隆索·邱奇), 790  
 Church–Turing thesis (邱奇-图灵论题)  
   extended (扩展), 823  
   overview (概述), 790–791  
   Turing machine simulation (图灵机模型), 798  
   virtual machines (虚拟机), 958  
 Ciphers, Kamasutra (Kamasutra 密码), 377  
 Ciphertext (密文), 993  
 Circuit models (电路模型)  
   building circuits (构建电路), 1006–1008  
   connections (连接), 1002–1004  
   controlled switches (控制开关), 1005–1006  
   conventions (惯例), 1004  
   inputs (输入), 1002–1004  
   logical design (逻辑设计), 1008–1009

- outputs (输出), 1002–1004
- overview (概述), 1002–1003
- wires (导线), 1002–1004
- Circuits (电路)
  - combinational (组合电路)
  - description (描述), 1010
  - from gates (门), 1019–1021
  - memory (内存), 1054–1057
- Circular linked lists (循环链表), 622
- Circular queues (循环队列), 620
- Circular shifts (循环位移), 375
- .class extension (.class 扩展), 3, 8, 228
- ClassDefFoundError, 160
- Classes (类), 4–5
  - accessing (访问), 227–229
  - description (描述), 226
  - implementing (实现), 383–389
  - inner (内部), 609
  - modules as (建模为), 228
  - variables (变量), 284
- Classifying NP-complete problems (NP 完全问题分析), 851
- Client code (客户端代码)
  - data types (数据类型), 430
  - library methods (库方法), 230
- Clocks (时钟)
  - CPU (中央处理器), 1077–1079
  - fetch and execute (取指并执行), 1059, 1061
  - overview (概述), 1058–1059
  - run and halt inputs (运行并停止输入), 1060
  - write control (写入控制), 1059–1060
- Closure operation in REs (RE 中的闭包操作), 724
- Clouds, plasma (等离子体云), 280
- Clustering coefficients (聚类系数)
  - global (全局), 713
  - local (局部), 693–694
- CMYK color format (CMYK 颜色格式), 48–49, 371
- Code and coding (代码与编程)
  - description (描述), 2
  - encapsulating (封装), 438
  - incremental development (增量开发), 319, 701
  - reuse (复用), 226, 253, 701
  - static methods (静态方法), 205–206
- Codebooks (密码本), 992
- Codons (密码子), genes (基因), 336
- Coefficients for floating-point numbers (浮点数的系数), 889
- Coercion (强制转换), 33
- Coin flip (抛掷硬币), 52–53
- Collatz function (克拉茨函数), 784
- Collatz problem (克拉茨问题), 296–297, 818
- Collatz sequence (克拉茨序列), 948
- Collections (集合)
  - description (描述), 566
  - iterable (可迭代), 601–605
  - objects (对象), 582–583
  - queues (队列)
  - stacks (栈)
  - symbol tables (符号表)
- Colons (冒号)
  - in Turing machine tapes (在图灵机纸带中), 767
  - foreach statements (foreach 语句), 601–602
- Color and Color data type (颜色和 Color 数据类型)
  - blobs, 709
  - compatibility (兼容性), 344
  - conversion (转换), 48–49
  - drawings (绘图), 150
  - grayscale (灰度), 344
  - luminance (亮度), 343
  - memory (内存), 514
  - overview (概述), 341–343
- Columns in 2D arrays (二维数组中的列), 106, 108
- Combinational circuits (组合电路)
  - adders (加法器), 1028–1030
  - ALUs (算术逻辑单元), 1031–1033
  - buses (总线), 1034–1036
  - decoders (解码器), 1021–1022
  - demultiplexers (多路分配器), 1022
  - description (描述), 1007–1008
  - gates (门), 1013–1021
  - layers of abstraction (抽象层), 1037–1039
  - modules (模块), 1034
  - multiplexers (多路复用器), 1023
  - overview (概述), 1012
  - sum-of-products (积之和), 1024–1027

- Comma-separated-value (.csv) files (逗号分隔值文件), 358, 360
- Command-line arguments (命令行参数), 7–8, 11, 127
- Commas (逗号)
  - arguments (参数), 30
  - constructors (构造函数), 333
  - lambda expressions (Lambda 表达式), 450
  - methods (方法), 30, 196
  - two-dimensional arrays (二维数组), 108
- Comments (注释), 5, 9
- Commercial data processing (商业数据处理), 410–413
- Common sequences (常见序列), longest (最长), 285–288
- Commutative axiom (交换律公理), 990
- Compact trace format (紧凑的跟踪格式), 770
- Comparable interface (可比较接口), 451, 545
- Comparable keys (可比较键)
  - sorting (排序), 546
  - symbol tables (符号表), 626–627
- CompareDocuments program (程序 CompareDocuments), 462–463
- compareTo() method (compareTo() 方法)
  - description (描述), 451
  - String (字符串), 332
  - user defined (用户定义), 545–546
- Comparisons arrays (可比较数组), 117
  - chained (链式), 43
  - objects (对象), 364, 545–546
  - operators (运算符), 27–29
  - performance (性能), 508–509
  - sketches (文档摘要), 462–463
- Compatibility (兼容性)
  - backward (向后), 976
  - Color (Color 类型), 344
- Compile-time errors (编译时错误), 6
- Compilers (编译器)
  - description (描述), 3, 589
  - optimizing (优化), 814
  - programs as data (程序作为数据), 922–924
  - purpose (目的), 788
  - TOY machine (TOY 机), 964–965
- Compiling (编译)
  - array values set at (数组值设置为), 95–96, 108
  - classes in (类), 229
  - description (描述), 2
  - programs (程序), 3
- Complement operation (补充操作)
  - bitwise (按位), 891
  - Boolean algebra (布尔代数), 990
- Complete small-world graphs (完成小世界图), 694
- Complex program (程序 Complex)
  - chaining method calls (链式方法调用), 404
  - encapsulation (封装), 433–434
  - instance variables (实例变量), 403–404
  - objects (对象), 404
  - overview (概述), 402–403
  - program (程序), 405
- Complex numbers (复数), 406–409
- Compound assignments (组合赋值), 60
- Compression (压缩), optimal (最优), 814
- Computability (可计算)
  - algorithms (算法), 787
  - halting problem (停机问题), 808–810
  - Hilbert's program (希尔伯特计划), 806–807
  - liar's paradox (说谎者悖论), 807–808
  - overview (概述) 806
  - reduction (归约), 811–813
  - unsolvability proof (不可证明的证明), 810
  - unsolvable problems (不可解问题)
- Computation: Finite and Infinite (计算: 有限机和无限机), 780
- Computational biology (计算生物学), 732–734
- Computational models (计算模型), 716
- Computer animations (计算机动画), 151
- Computer speed in performance (计算速度的性能表现), 507–508
- Computer systems (计算机系统), 1094–1095
- Computers and Intractability A Guide to the Theory of NP Completeness book (计算机与难解问题: NP 完全性理论指南), 859
- Computers Ltd: What They Really can't Do book (计算机的极限: 它们真正不能办到的事), 780
- Computing devices (计算设备)
  - boolean logic (布尔逻辑)
  - circuit models (电路模型)

- combinational circuits (组合电路)
- digital (数字)
- overview (概述), 985
- sequential circuits (时序电路)
- Computing machines (计算机器)
  - machine-language programming (机器语言编程)
  - overview (概述), 873
  - representing information (表示信息)
- TOY (TOY 机)
- Computing sketches (计算草图), 459–460
- Concatenation (拼接)
  - files (文件), 356
  - strings (字符串), 19–20, 723–724
- Concert A, 155
- Concordances (词汇索引), 659
- Conditionals and loops (条件和循环), 50
  - applications (应用), 64–73
  - break statement (break 语句), 74
  - continue statement (continue 语句), 74
  - do-while loops (do-while 循环), 75
  - examples (示例), 61
  - for loops (for 循环), 59–61
  - if statement (if 语句), 50–53
  - infinite loops (死循环), 76
  - miscellaneous (杂项), 74–75
  - in modular programming (在模块化编程中), 227–228
  - nesting (嵌套), 62–64
  - performance analysis (性能分析), 500, 510
  - static methods (静态方法), 193–195
  - summary (总结), 77
  - switch statement (switch 语句), 74–75
  - TOY machine (TOY 机), 913, 918–921
  - while loops (while 循环), 53–59
- Connected components (连接组件), 709
- Connecting programs (连接程序), 141
- Connections (连接)
  - buses (总线), 1034
  - circuit models (电路模型), 1002–1004
  - CPU (中央处理器), 1077
  - power source (电源), 1003–1004
  - program counters (程序计数器), 1073–1075
- Constant order of growth (常量增长量级), 503
- Constants (常量), 16
- Constructors (构造函数)
  - data types (数据类型), 384–385
  - String (字符串), 333
- Containing symbol table keys (包含符号表键), 624
- Context-free languages (无上下文的语言), 755
- Continue statements (continue 语句), 74
- Contracts (契约)
  - APIs (应用程序编程接口), 230–231
  - design by contract (契约式设计), 465–467
  - interface (接口), 446–447
  - machine-language (机器语言), 932
- Control characters (控制字符), 894
- Control circuit (控制电路)
  - CPU (中央处理器), 1078
  - execute signals (执行信号), 1082–1083
  - fetch signals (取指信号), 1080, 1082–1083
  - overview (概述), 1080
- Control flow (控制流程)
  - conditionals and loops (条件和循环)
  - static method calls (静态方法调用), 193–195
- Control lines (控制线)
  - CPU (中央处理器), 1077–1080
  - memory bits (内存位), 1056
  - multiplexers (多路复用器), 1019–1020
  - program counters (程序计数器), 1074–1075
  - register bits (寄存器位), 1051
- Controlled switches (控制开关), 1002–1003, 1005–1006
- Conversion codes (转换代码), 131–132
- Conversion specifications (转换规范), 130–131
- Conversions (转换)
  - casts (转型), 33–34
  - color (颜色), 48–49
  - data types (数据类型), 339
  - decimal to binary (十进制转二进制), 877
  - explicit (显式地), 34–35
  - hex and binary (十六进制和二进制), 876–877
  - implicit (隐式地), 33
  - numbers (数字), 21, 67–69
  - overview (概述), 32
  - strings (字符串), 21, 453, 880–881
- Convert program (程序 Convert), 880–882
- Conway, John (约翰·康威), 326, 794

Cook, Stephen, (库克·史蒂芬) 840, 845  
 Cook-Levin theorem (库克·莱维定理), 844–845, 847  
 Cook reduction (库克归约), 841  
 Coordinates (坐标)  
   drawing (绘图), 144–146  
   images (图像), 347  
   polar (极), 47  
 Corner cases (边界条件), 236  
 Cosine similarity measure (余弦相似性度量法), 462  
 Cost of immutable types (不可变类型的代价), 440  
 Coulomb's law (库仑定律), 383  
 Counter circuits (计数器电路), 1008  
 Counter machines (计数器), 794  
 Counter program (计数程序), 436–437  
 Coupon collector problem (卡券收集问题), 101–103  
 Coupon program (程序 Coupon), 206  
 CouponCollector program (程序 CouponCollector), 101–103, 205  
 CPUs (中央处理器)  
 Craps game (掷骰子游戏), 259  
 Cray, Seymour (西摩·格雷), 971  
 Crichton, Michael (迈克尔·克莱顿), 424  
 Cross-coupled NOR gates (交叉耦合或非门), 1050  
 Cross-coupled switches (交叉耦合开关), 1049  
 Cross products of vectors (向量交叉乘积), 472  
 Cryptographic keys (加密密钥), 992  
 Cryptography application (密码学应用), 992–994  
 Cryptosystems (加密系统), 992–993  
 <Ctrl-C> keys (<Ctrl-C> 键), 76  
 <Ctrl-D> keys (<Ctrl-D> 键), 137  
 <Ctrl-Z> keys (<Ctrl-Z> 键), 137  
 Cubic order of growth (三次型增长量级), 505–508  
 Cumulative distribution function (累计分布函数), 202–203  
 Curly braces ({} )  
   regular expressions (正则表达式), 724, 731  
   statements (语句), 5, 78–79  
   static methods (静态方法), 196  
   two-dimensional arrays (二维数组), 108  
 Curves (曲线)

Brownian bridges (布朗桥), 278–280  
 Dragon (龙形), 49, 424  
 Koch (科赫), 397  
   space-filling (空间填充), 425  
 spirals (螺旋), 398–399  
 Cycles per second (每秒周期数), 155

## D

Dantzig, George (乔治·丹齐格), 831  
 Data abstraction (数据抽象), 329, 382  
 Data as instructions (数据作为指令), 922–924  
 Data compression (数据压缩), 814  
 Data-driven code (数据驱动代码), 141, 171, 184  
 Data mining example (数据挖掘示例), 458–459  
 Data paths for buses (总线数据路径), 1034  
 Data structures (数据结构), 493  
   arrays (数组)  
   binary search trees (二叉搜索树)  
   linked lists (链表), 571–578  
   queues (队列)  
   resource allocation (资源分配), 606–607  
   stacks (栈)  
   stock example (股票示例), 411  
   summary (总结), 608  
   symbol tables (符号表)  
 Data-type design (数据类型设计)  
   APIs (应用程序编程接口), 429–43  
   data mining example (数据挖掘示例), 458–464  
   design by contract (按契约设计), 465–467  
   encapsulation (封装), 432–438  
   immutability (不可变), 439–446  
   subclassing (子类化), 452–457  
   subtyping (子类型化), 446–45  
   overview (概述), 428  
 Data types (数据类型)  
   access modifiers (访问修饰符), 38  
   APIs (应用程序编程接口), 38  
   boolean, 99  
   built-in (内置), 38  
   Color, 341–34  
   Complex, 402–405  
   constructors (构造函数), 384–385  
   conversions (转换), 34–35, 339  
   creating (创建), 38

- definitions (定义), 331–335
- DrunkenTurtle, 400–401
- elements summary (要素摘要), 383
- generic (通用), 583–58
- Histogram, 392–39
- image processing (图像处理), 346–352
- immutable (不可变), 364, 43
- input and output (输入和输出), 353–362
- insertion sorts (插入排序), 545–548
- instance methods (实例方法), 385–386
- instance variables (实例变量), 384
- Koch, 397
- Mandelbrot, 406–409
- output (输出), 355
- overview (概述), 330
- reference (引用), 362–369
- Spiral, 398–399
- StockAccount, 410–413
- Stopwatch, 390–391
- String
  - summary (总结), 368
- TOY machine (TOY 机), 907
- Turtle, 394–396
- type safety (类型安全), 18
- variables within methods (方法中的变量), 386–388
- Data visualization (数据可视化), 307–309
- Davis, Martin (马丁·戴维斯), 816
- Dead Sea Scrolls (死海古卷), 659
- Debugging (调试)
  - abstraction layers (抽象层), 1037
  - assertions (断言), 466–467
  - encapsulation for (封装), 432
  - immutable types (不可变类型), 440
  - incremental (增量), 317, 319
  - linked lists (链表), 596
  - modular programming (模块化编程), 251–254
  - test client main() for (测试客户端 main()), 235
  - unit testing (单元测试), 246
- Decidability (可判定性), 786–787
- Decimal number system (十进制系统)
  - conversion to binary (转化为二进制), 877
  - description (描述), 38, 875
  - examples (示例), 878–879
- Decision problems (决策问题), NP, 835
- Decks of cards (一副扑克), 97–100
- Declaration statements (声明语句), 15–16
- Declaring (声明)
  - arrays (数组), 91, 116
  - String variables (字符串变量), 333
- Decoders (解码器), 1021–1022
- Decoding numbers (解码数字), 889
- Decrementers (递减器), binary (二进制), 770–771
- Decryption devices (解密设备), 992
- Dedup operation (除尘操作)
  - punched paper tape (穿孔纸带), 942–944
  - strings (字符串), 652–653
- DeDup program (程序 DeDup), 652–653
- Default values (默认值)
  - arrays (数组), 93, 106–107
  - canvas size (画布尺寸), 145
  - ink color (油墨颜色), 150
  - instance variables (实例变量), 415
  - Node objects (Node 对象), 572
  - pen radius (笔半径), 146
- Defensive copies (防御拷贝), 441
- Defining (定义)
  - functions (函数), 192
  - interfaces (接口), 446
  - static methods (静态方法), 193, 196
- Definite integration (定积分), 816
- Degrees of separation (分隔度)
  - description (描述), 670
  - shortest paths (最短路径), 684–686
- DeMorgan's laws (德·摩根定律), 989–991, 1014–1015
- Demultiplexers (多路分配器), 1022
- Denial-of-service attacks (拒绝服务攻击), 512
- Dependencies in subclasses (子类中的依赖关系), 453
- Dependency graphs (依赖图), 252
- Deprecated methods (不推荐使用的方法), 469
- Depth-first searches (深度优先搜索算法)
  - vs. breadth-first searches (广度优先搜索算法), 690
  - percolation case study (渗透案例研究), 312
- Deque (双端队列), 618
- Derived classes (派生类), 452

Descartes, René (勒内·笛卡儿), 398

Design (设计)

APIs (应用程序编程接口), 233

by contract (按照契约), 465–467

data types (数据类型)

Deterministic finite-state automata (DFAs) (确定有限状态自动机)

examples (示例), 740–741

implementation (实施), 741–743

Kleene's theorem (克林定理)

language recognized (识别的语言), 739–740

NFA equivalence (NFA 等价), 749–750

nondeterminism (非确定性), 744–748

operations (操作符), 739

overview (概述), 738

power limitations (功率限制), 753–755

summary (总结), 756

universal virtual (通用虚拟), 788–789

DFA program (程序 DFA), 742–743

Diameters of graphs (图的直径), 711

Diamond operators ( $\diamond$ ) (钻石运算符), 585

Dice (骰子)

Sicherman (塞克文骰子), 259

simulation (模拟), 121

Dictionary lookup (字典查找), 624, 628–632

Difficult problems (困难问题)

intractability (难以处理), 828–829

search problems (搜索问题), 837–838

Digital circuits (数字电路), 1013

Digital devices (数字设备), 1070

control (控制), 1080–1082

CPU (中央处理器), 1076–1080

program counters (程序计数器), 1073–1075

Digital image processing (数字图像处理)

digital images (数字图像), 346–347

fade effect (淡入淡出效果), 351–352

grayscale (灰度), 347–349

overview (概述), 346

scaling (缩放), 349–350

Digital signal processing (数字信号处理), 155, 158

Dijkstra, Edsger (埃德加·迪克斯特拉)

Dutch-national-flag problem (荷兰国旗问题), 564

goto statements (goto 语句), 926

two-stack algorithm (双栈算法), 587

Dijkstra's algorithm (Dijkstra 算法), 692

Diophantine (丢番图), 816

Directed graphs (有向图), 711

Directed percolation (有向渗透原理), 317

Discrete distributions (离散分布), 172

Disjunctive normal forms (析取范式), 996–997

Distances of graph paths (图路径的距离), 683, 687–688

Distributive axiom (分配律公理), 990

Divide-and-conquer approach (分治策略)

linearithmic order of growth (线性增长顺序), 504

mergesort (归并排序算法), 550–551, 554

Division (除法)

floating-point numbers (浮点数), 24–26

integers (整数), 22–23

polar representation (极坐标表示), 433

DivisorPattern program (程序 DivisorPattern), 62–64

DNA computers (DNA 计算机), 795

DNS (domain name system) (域名系统), 629

do-while loops (do-while 循环), 75

Documents, searching for (文档搜索), 464

Dollar signs (\$) in Res (正则表达式中的 \$ 符号), 731

Domain name system (DNS) (域名系统), 629

Domains, function (域名函数), 192

Dot products (点积)

function implementation (函数实现), 209

vectors (向量), 92, 442–443

Double.parseDouble() method (Double.parseDouble() 方法)

calls to (调用), 30–31

type conversion (类型转换), 21, 34

Double buffering drawings (双缓冲绘图), 151

double data type (double 数据类型)

conversion codes (转换代码), 132

description (描述), 14–15

input (输入), 133

memory size (内存大小), 513

overview (概述), 24–26

Double negation identity (双重否定等式), 990



Double negatives in gates (双重否定门), 1015–1016

Double quotes ("")(双引号)

escape sequences (转义序列), 19

text (文本), 5, 10

Doublet game (偶极子游戏), 710

Doubling hypotheses (倍增假说), 496, 498–499

DoublingTest program (程序 DoublingTest), 496, 498–499

Downscaling in image processing (缩小图像处理), 349

Dragon curves (龙形曲线), 49, 424

Dragon program (程序 Dragon), 163

Draw library (画图库), 361

Drawings (绘图)

recursive graphics (递归图形), 276–277

standard (标准)

DrunkenTurtle program (程序 DrunkenTurtle), 400

DrunkenTurtles program (程序 DrunkenTurtles), 401

Dumping virtual machines (转储虚拟机), 960–961

Dutch-national-flag problem (荷兰国旗问题), 564

Dynamic dictionaries (动态词典), 628

Dynamic dispatch (动态调度), 448

Dynamic programming (动态编程)

bottom-up (自下而上), 285

longest common subsequence (最常见的子序列), 285–288

overview (概述), 284

summary (总结), 289

top-down (自顶向下), 284

## E

Easy problems (简单问题)

intractability (难处理性), 829

search (搜索), 837

Eavesdroppers (窃听者), 992–993

Eccentricity in vertices (顶点的偏心距), 711

Eckert, J. Presper (普雷斯伯·埃克特), 924–925

Edges (边)

graphs (图), 671, 674

self-loops and parallel (自循环和平行), 676

EDVAC computer (EDVAC 计算机), 924–925

Efficiency (效率)

n-body simulation (多体模拟), 488

random web surfer (随机网络冲浪), 185

Turing machines (图灵机), 772

Efficient algorithms (有效算法), 532

Einstein, Albert (阿尔伯特·爱因斯坦), 400

Election voting machine errors (选举投票机错误), 436

Electric charge (电荷), 383–389

Element distinctness problem (元素不同性问题), 554

Elements in arrays (数组中的元素), 90

else clauses (else 子句), 51–52

Empirical analyses (实证分析), 496–497

Empty strings with REs (带有正则表达式的空字符串), 724

Emulators (仿真器), 965

Encapsulation (封装)

code clarity (代码简洁性), 438

error prevention (错误预防), 436–437

example (示例), 433–434

modular programming (模块化编程), 432

overview (概述), 432

planning for future (未来规划), 435

private access modifier (私有访问修饰符), 433

Encoding numbers (编码), 889

Encryption devices (加密设备), 992

End-of-file sequence (文件结束序列), 137

Enhancements for Turing machines (图灵机的增强功能), 792–793

ENIAC computer (ENIAC 计算机), 924–925

Enigma code (英格玛密码), 717

Entropy (熵)

Shannon (香农), 378

text corpus (文本语料库), 667–668

Equals signs(=)

assignment statements (赋值语句), 17

assignment vs. boolean (赋值与布尔值), 42, 78

comparisons (比较), 27–29, 364

compound assignments (组合赋值), 60

vs. equals() (与 equals() 比较), 369–370

Equality of objects (对象相等性), 364, 454–456

equals() method (equals() 方法)

Color (颜色), 343

vs. equals signs (等号), 369–370

Object (对象), 453–455

- String (字符串), 332
  - Equilateral triangles (等边三角形), 144–145
  - Equivalence problem for Res (正则表达式等价问题), 728
  - Equivalent models for Turing machines (图灵机的等效模型), 792–793
  - Erdős, Paul, 686
  - Erdős–Renyi model (Erdős–Renyi 图模型), 695, 712
  - Errors (错误)
    - aliasing (别名), 363
    - debugging (调试)
    - encapsulation for (封装), 436–437
    - overview (概述), 6
    - syntax (语法), 10–11
    - testing for (测试), 318
  - Escape characters (转义字符), 730, 757
  - Escape sequences (转义序列), 19
  - Euclidean distance (欧氏距离)
    - sketch comparisons (摘要比较), 462–463
    - vectors (向量), 118
  - Euclid's algorithm (欧几里得算法)
    - description (描述), 85
    - machine-language (机器语言), 931
    - recursion (递归), 267–268
    - TOY machine (TOY 机), 918–921
  - Euler, Leonhard (莱昂哈德·欧拉), 89
  - Euler's constant (欧拉常数), 222
  - Euler's sum-of-powers conjecture (欧拉幂之和猜想), 89
  - Euler's totient function (欧拉函数), 222
  - Evaluate program (求值程序), 588–589
  - Evaluating expressions (求值表达式), 17, 586–589
  - Event-based programming (基于事件的编程), 451
  - Exception class (异常类), 465
  - Exception filters (异常过滤器), 540
  - Exceptions (异常), 465–467
  - Exchanging values (交换值)
    - arrays (数组), 96
    - function implementation (函数实现), 209
  - Exclamation points (!)
    - not operator (否操作符), 26–27, 991
    - comparisons (比较), 27–29
  - Exclusive or operation (异或运算符)
    - bitwise (按位), 891–892, 913
    - boolean (布尔), 987–989
    - sum-of-products (积之和), 1024–1025
  - Execute phase in CPU (CPU 执行阶段), 1079
  - Explicit casts (显式转型), 33–34
  - Exponential distributions (指数分布), 597
  - Exponential order of growth (指数增长量级), 505
    - difficult problems (难题), 828–829
    - intractability (难解性), 826
    - overview (概述), 272–273, 506
    - playing card possibilities (扑克牌可能性), 823
    - running time (运行时间), 507–508
    - SAT problem (SAT 问题), 856
    - usefulness (有效性), 858
  - Expressions (表达式)
    - arithmetic evaluation (算术运算), 586–589
    - boolean (布尔), 995–996
    - description (描述), 17
    - Lambda, 450
    - method calls (方法调用), 30
    - regular (正则)
  - Extended Church–Turing thesis (扩展邱奇–图灵论题), 823
  - Extensible libraries (可扩展库), 452
  - ExtractFloat program (程序 ExtractFloat), 893
  - Extracting data (抽取数据), 358, 360
- ## F
- Factor problem (因式分解问题), 838, 859
  - Factorials (因子), 264–265
  - Factoring (分解), 72–73, 827, 838
  - Factors program (程序 Factors), 72–73
  - Fade effect (淡入淡出效果), 351–352
  - Fade program (程序 Fade), 351–352
  - Fair coin flip (公平硬币翻转), 52–53
  - Falsifiable hypotheses (证伪的假说), 495
  - Fecundity parameter (繁殖率参数), 89
  - Feedback circuits (回馈电路), 1048–1049
  - Fermat's Last Theorem (费马大定理), 89, 722
  - Ferns, Barnsley (巴恩斯利蕨), 240–243
  - Fetch-increment-execute cycle (取指–递增–执行周期), 910–911

## Fibonacci numbers (斐波那契数列)

- formulas (公式), 82
- machine language (机器语言), 935–936
- recursion (递归), 282–283

## FIFO queues (FIFO 队列)

## Files (文件)

- concatenating and filtering (拼接和过滤), 356
- format (格式), 237
- in I/O (在 I/O 中), 126
- n-body simulation (多体模拟), 483
- redirection (重定向), 139–141
- splitting (分裂), 360
- stock example (股票示例), 411
- symbol tables (符号表), 629

## Filled shapes (填充形状), 149

## Filters (过滤器)

- exception (异常), 540
- files (文件), 356
- image processing (图像处理), 379
- pipng (管道), 142–143
- standard drawing data (标准绘图数据), 146–147
- standard input (标准输入), 140

## final keyword (final 关键字)

- description (描述), 384
- immutable types (不可变类型), 440
- instance variables (实例变量), 404

## Financial systems, graphs for (金融系统、图表), 673

## Finite-state transducers (有限状态传感器), 762

## Finite sums (有限和), 64–65

## First-in first-out (FIFO) queues (先进先出队列)

- applications overview (应用概述), 597
- array implementation (数组实现), 596
- linked-list implementation (链表实现), 593
- M/M/1 (M/M/1 队列), 597–600
- overview (概述), 566, 592–593

## Flexibility (灵活性), 702

## Flip program (程序 Flip), 52–53

## Flip-flops (触发器), 1049–1050

- float data type (浮点数据类型), 26, 513

## Floating-point numbers (浮点数)

- conversion codes (转换代码), 131–132
- exponents (指数), 889

## overview (概述), 24–2

## precision (预测), 40

## representing (表示), 888–890

## storing (存储), 40

## Flow of control (控制流)

## conditionals and loops (控制和循环)

## static method calls (静态方法调用), 193–195

## Flowcharts (流程图), 51–52

## for loops (for 循环)

## continue statement (continue 语句), 74

## examples (示例), 61

## nesting (嵌套), 62–64

## working with (共同作用), 59–61

## Foreach statements (foreach 语句), 601–602

## Formal languages (形式语言)

## abstract machines (抽象机), 737–738

## alphabets (字母表), 720–721

## binary strings (二进制字符串), 718–719

## definitions (定义), 718–723

## DFAs (确定有限状态自动机)

## recognition problem (识别问题), 722

## regular (正则), 723–729

## regular expressions (正则表达式)

## specification problem (规范问题), 722

## Format, files (格式文件), 237

## Format strings (格式化字符串), 130–131

## Formatted input (格式化输入), 135

## Formatted printing (格式化打印), 130–132

## Forth language (Forth 语言), 590

## Fortran language (Fortran 语言), 1094

## Fourier series (傅里叶级数), 211

## Fractal dimensions (分形维度), 280

## Fractals (分形), 278–280

## Fractional Brownian motion (分形布朗运动), 278

## Fractions (分数), 889–890

## Fragile base class problem (脆弱的基类问题), 453

## Freeing memory (释放内存), 367

## Frequencies (频率)

## counting (计数), 555

## sorting (排序), 556

## Zipf's law (齐夫定律), 556

## FrequencyCount program (程序 FrequencyCount), 555–557

Fully parenthesized arithmetic expressions (带完全括号的算术表达式语句), 587

Function calls (函数调用)

abstraction (抽象), 590–591

static methods (静态方法), 197

traces (跟踪), 195

trees (树), 269, 271

Function graphs (函数图), 148, 248

Functional interfaces (函数式接口), 450

Functional programming (函数化编程), 449

Functional property of programs (程序的函数属性), 812–813 Functions

boolean (布尔), 987–991, 994–997

computing with (计算), 449

defining (定义), 192

inverting (反向), 536–538

iterated function systems (迭代函数系统), 239–243

libraries (库)

machine language (机器语言), 931–933

mathematical (数学), 202–204

modules (模块)

overview (概述), 191

recursive (递归)

static methods (静态方法), 193–201

tables of (表), 907–908

## G

Gambler program (Gambler 程序), 70–71

Gambler's ruin simulation (赌徒破产问题), 69–71

Game of Life (生命游戏), 326, 794

Garbage collection (垃圾回收), 367, 516

Gardner, Martin (马丁·加德纳), 424

Garey, Michael R. (迈克尔·加里), 859

Gates (门)

abstraction layers (抽象层), 1037

AND (与运算符), 1014

circuits from (电路), 1019–1021

multiway (多路), 1015–1017

NOR (或非), 1014

NOT (非), 1013–1014

OR (或), 1014

overview (概述), 1013

sum-of-products (积之和), 1026–1027

summary (总结), 1018–1019

universal sets of (通用集合), 1045

Gaussian distribution functions (高斯分布函数)

API (应用程序编程接口), 231

cumulative (累计), 202–203

probability density (概率密度), 202–203

Gaussian elimination (高斯消元), 830

Gaussian program (程序 Gaussian), 203

Gaussian random numbers (高斯随机数), 47

General purpose computers (通用计算机), 790

Generalized multiway gates (广义多路门), 1016–1017

Generalized regular expressions (广义正则表达式), 730–732

Generic types (通用类型), 583–585

Genomics (基因组学)

application (应用), 336–340

indexing (索引), 634

regular expressions (正则表达式), 727, 732–734

symbol tables (符号表), 629

Geometric mean (几何平均数), 162

Geometry (几何)

abstraction layers (抽象层), 1037–1039

gates (门), 1015–1016

German Enigma code (德国英格玛密码), 717

Get operations (Get 操作)

hash tables (散列表), 639

symbol tables (符号表), 624

Gilbert–Shannon–Reeds model (Gilbert–Shannon–Reeds 模型), 125

Glass filters (毛玻璃过滤器), 379

Global clustering coefficients (全局聚类系数), 713

Global variables (全局变量), 284

Glossary of terms (术语表), 1097–1101

Gödel, Kurt (库尔特·哥德尔), 807, 840

Golden ratio (黄金比例), 83

Goldstine, Herman (赫尔曼·戈德斯坦), 925

Gore, Al (阿尔·戈尔), 436

Gosper, R. (高斯帕), 805

Goto statements (goto 语句), 926

Graph data type (图数据类型), 675–679

Graph program (程序 Graph), 676–679

**Graphics (图形)**

- recursive (递归), 276–277, 397
- turtle (乌龟), 394–396

**Graphs (图)**

- bipartite (二分图), 682
- client example (客户端示例), 679–682
- connected components (连接组件), 709
- dependency (依赖性), 252
- description (描述), 671
- DFA's (确定有限状态自动机), 738
- diameters (直径), 711
- directed (有向), 711
- examples (示例), 695
- function (函数), 148, 248
- generators (生成器), 700
- Graph data type (图数据类型), 675–679
- grid (网格), 708
- isomorphism problem (同构问题), 859
- lessons (经验总结), 700–702
- matching (匹配), 713
- overview (概述), 670–671
- random web surfer (随机网络冲浪), 170
- small-world (小世界), 693–699
- systems examples (系统示例), 671–674
- vertex cover (顶点覆盖), 828, 834, 842

**Gravity (重力), 481****Gray codes (格雷码), 273–275****Grayscale (灰度图)**

- Color (颜色), 344
- image processing (图像处理), 347–349

**Grayscale program (灰度图程序), 347–349****Greater than signs (>)**

- bitwise operations (按位运算), 891–892
- comparisons (比较), 27–29
- lambda expressions (Lambda 表达式), 450
- redirection (重定向), 139–140

**Greatest common divisor (gcd)(最大公约数)**

- machine language (机器语言), 931
- recursive algorithm (递归算法), 267–268
- TOY machine (TOY 机), 918–921

**Grep program (程序 Grep), 736****grep tool (grep 工具)**

- filters (过滤器), 142–143
- regular expressions (正则表达式), 734–736

**Grid graphs (网格图), 708****Guarantees (保证)**

- NP-complete problems (NP 完全问题), 852
- performance (性能), 512, 627
- worst-case analysis (最坏情况分析), 825

**H****H-trees of order n (n 阶 H 树), 276–277****Hadamard matrices (哈达玛矩阵), 122****Halt instructions (停机指令)**

- CPU (中央处理器), 1079
- TOY machine (TOY 机), 912

**Halting problem (停机问题), 808–810****Hamilton, William (威廉·汉密尔顿), 424****Hamming distances (汉明距离), 295****Handles for pointers (指针句柄), 371****Hardy, G. H. (戈弗雷·哈罗德·哈代), 86****Harel, David (大卫·哈尔), 780****Harmonic mean (调和平均值), 162****Harmonic numbers (谐波数)**

- finite sums (有限和), 64–65
- function implementation (函数实现), 199

**Harmonic program (程序 Harmonic), 193–195****HarmonicNumber program (程序 HarmonicNumber), 64–65****Harmonics and chords (和声和和弦), 211****Hash codes and hashing operation (散列代码和散列操作)**

- object equality (对象相等), 454–455
- sketches (文档摘要), 460
- strings (字符串), 515
- symbol tables (符号表), 624

**Hash functions (散列函数), 636****Hash tables (散列表), 636–639****Hash values (散列值), 636****Hashable keys (可散列的键), 626****hashCode() method (hashCode() 方法)**

- Object (对象), 453, 455–456
- String (字符串), 332

**HashMap class (HashMap 类), 655****HashST program (程序 HashST), 637–638****Heap memory (堆内存), 516****Heap-ordered binary trees (堆有序的二叉树), 661****Height in binary search trees (二叉搜索树的高度),**

- 640
- HelloWorld program (程序 HelloWorld), 4–6
- Hertz (赫兹), 155
- Hexadecimal (hex) notation (十六进制表示法)
- conversions with binary (转换为二进制), 876–877
  - description (描述), 875–876
  - examples (示例), 878–879
  - literals (常量), 891
  - memory (内存), 909
- Hilbert, David (戴维·希尔伯特), 425, 806, 816
- Hilbert curves (希尔伯特曲线), 425
- Hilbert's 10th problem (希尔伯特十大问题), 816
- Hilbert's program (希尔伯特程序), 806–807
- Histogram program (直方图程序), 392–393
- Histograms (直方图), 177
- Hoare, C. A. R. (查尔斯·安东尼·理查德·霍尔爵士), 518
- Hollywood numbers (好莱坞数), 711
- Horner, William (威廉·霍纳), 957
- Horner's method (霍纳方法), 223, 882, 956–957
- Htree program (程序 Htree), 276–277
- Hurst exponent (赫斯特指数), 280
- Hyperbolic functions (双曲函数), 256
- Hyperlinks (超链接), 170
- Hypotenuse of right triangles (直角三角形的斜边), 199
- Hypotheses (假说)
- doubling (倍增), 496, 498–499
  - falsifiable (可证伪的), 495
  - mathematical analysis (数学分析), 498, 500–502
  - overview (概述), 496
- I
- I/O (输入输出)
- Identifiers (标识符), 15–16
- Identities (标识)
- Boolean algebra (布尔代数), 989–990
  - exclusive or function (异或函数), 993
  - objects (对象), 338, 340
- IEEE 754 standard (IEEE 754 标准), 40, 888–889
- if statements (if 语句)
- nesting (嵌套), 62
  - working with (共同作用), 50–53
- IFS program (IFS 程序), 241, 251
- IllegalFormatConversionException, 131
- ILP problem (integer linear programming problem) (ILP 问题, 整数线性规划问题), 831
- NP-completeness (NP 完全性), 846
  - vertex cover problem (顶点覆盖问题), 842
- Immutable types (不可变类型), 364, 439
- advantages (优点), 440
  - arrays and strings (数组与字符串), 439–440
  - cost (代价), 440
  - example (示例), 442–445
  - final modifier (final 修饰符), 440
  - references (引用), 441
  - symbol table keys (符号表键), 625, 655
- Implementation (实现)
- API methods (API 方法), 231
  - interfaces (接口), 447
- Implements clause (实现子句), 447
- Implicit type conversions (隐式类型转换), 33
- In data type (内置数据类型), 354–356
- Incremental development (增量开发), 319, 701
- Incrementers, binary (二进制递增器), 769–771
- Index program (程序 Index) 632–634
- IndexGraph program (程序 IndexGraph), 680–682
- Indexing (索引)
- arrays (数组), 90, 116
  - String (字符串), 332
  - symbol tables (符号表), 624, 632–634
  - zero-based (零基, 从 0 开始), 92
- Induced subgraphs (导出子图), 705
- Induction (归纳)
- mathematical (数学), 262, 266
  - recursion step (递归步骤), 266
- Infinite loops (无限循环), 76, 808–812
- Infinite tape for Turing machines (用于图灵机的无限纸带), 769, 774
- Infinity value (无限值), 26, 40
- Information content of strings (字符串的信息内容), 378
- Information representation (信息表示)
- Inheritance (继承)
- multiple (倍数), 470
  - subclassing (子类化), 452–457
  - subtyping (子类型), 446–451

- Initialization array (初始化数组), 93
  - inline (内联), 18
  - instance variables (实例变量), 415
  - two-dimensional array (二维数组), 106–107
- Inline variable initialization (内联变量初始化), 18
- Inner classes (内置类), 609
- Inner loops (内循环)
  - description (描述), 62
  - performance (性能), 500, 510
- Inorder tree traversal (有序树遍历), 649
- Input (输入)
  - arithmetic logic units (算术逻辑单元), 1031
  - array libraries (数组库), 237–238
  - circuit models (电路模型), 1002–1004
  - clocks (时钟), 1060
  - command-line arguments (命令行参数), 7
  - data types (数据类型), 353
  - demultiplexers (多路分配器), 1022
  - file concatenation (文件连接), 356
  - gates (门), 1013
  - insertion sorts (插入排序), 548–549
  - machine-language (机器语言), 936–938
  - multiplexers (多路复用器), 1019–1020
  - overview (概述), 126–129
  - in performance (性能), 510
  - program counters (程序计数器), 1073–1075
  - random web surfer (随机网络冲浪), 171
  - screen scraping (屏幕抓取), 357–359
  - standard (标准), 132–138
  - stream data type (流数据类型), 354–355
  - virtual machines (虚拟机), 969–970
- Input/off switches (输入 / 关闭开关), 1005
- InputMismatchException, 135
- Inserting (插入)
  - BST nodes (BST 节点), 644–645
  - linked list nodes (链表节点), 573–574
- Insertion program (程序 Insertion), 546–547
- Insertion sorts (插入排序)
  - data types (数据类型), 545–548
  - input sensitivity (输入敏感), 548–549
  - overview (概述), 543–544
  - performance (性能), 544–545
- InsertionDoublingTest
  - program (程序), 548–549
- Instance methods (实例方法)
  - data types (数据类型), 385–386
  - invoking (调用), 334
  - vs. static (静态), 340
- Instance variables (实例变量)
  - Complex program (程序 Complex), 403–404
  - data types (数据类型), 384
  - initial values (初始值), 415
- Instances of objects (对象实例), 333
- Instruction register (IR)(指令寄存器), 910
- Instructions (说明)
  - components (组件), 911
  - CPU (中央处理器), 1079–1080
  - as data (作为数据), 922–924
  - execution time (执行时间), 509
  - instruction sets (指令集), 911–913
  - parsing (解析), 966–967
  - TOY machine (TOY 机), 909
  - TOY-8 machine (TOY-8 机), 1070–1071
- Integer linear inequality (整数线性不等式)
  - satisfiability (可满足性), 831, 838, 845
- Integer linear programming (整数线性编程), 831
- NP-completeness (NP 完全性), 846
  - vertex cover problem (顶点覆盖问题), 842
- Integer.parseInt() method (Integer.parseInt() 方法)
  - calls to (调用), 30–31
  - type conversion (类型转换), 21, 23, 34
  - strings (字符串), 880–882
- Integers and int data type (整数和 int 数据类型)
  - arithmetic (算术), 884–885
  - bitwise operations (按位操作), 891–892
  - conversion codes (转换代码), 131–132
  - description (描述), 14–15
  - input (输入), 133–134
  - overview (概述), 22–24
- Integrals, approximating (近似积分), 449
- Integrated development environments (IDEs) (集成开发环境), 3
- Integration, definite (定积分), 816
- Interactions between modules (模块之间的交互), 319
- Interactive user input (交互式用户输入), 135–136
- Interface construct (接口构造), 446
- Interfaces (接口)



- APIs (应用程序编程接口), 430
  - built-in (内置), 451
  - circuit models (电路模型), 1003
  - CPU (中央处理器), 1076
  - defining (定义), 446
  - functional (泛函), 450
  - gates (门), 1016–1017
  - implementing (实现), 447
  - memory (内存), 1054
  - multiplexers (多路复用器), 1020
  - program counters (程序计数器), 1073
  - using (使用), 447–448
  - Internet DNS (互联网 DNS), 629–630
  - Internet Protocol (IP)(互联网协议), 435
  - Interpolation in fade effect (渐变效果中的插值), 351
  - Interpreters (解释器)
    - Evaluate program (程序 Evaluate), 589
    - TOY machine (TOY 机), 964
  - IntOps program (程序 IntOps), 23
  - Intractability (难解性)
    - difficult problems (难问题), 828–829
    - easy problems (简单问题), 829
    - exponential-time algorithms (指数时间算法), 826
    - main question (主要问题), 840–841
    - NP-completeness (NP 完全性)
    - numbers (数字), 827
    - overview (概述), 822–824
    - path problems (路径问题), 829
    - polynomial-time algorithms (多项式时间算法), 825–826
    - polynomial-time reductions (多项式时间归纳), 841–843
    - problem size (问题规模), 824
    - satisfiability (可满足性), 830–832
    - search problems (搜索问题), 833–840
    - subset sum problem (子集和问题), 827–828
    - vertex cover (顶点覆盖), 828
    - worst case (最坏情况), 825
  - Introduction to the Theory of Computation book(计算机理论入门书), 780
  - Invariants in assertions (断言中的不变量), 467
  - Inverse permutations (逆排序), 122
  - Inverters (反向), 1013–1014
  - Inverting functions (反向函数), 536–538
  - Invoking instance methods (调用实例方法), 334
  - IP (Internet Protocol)(互联网协议), 435
  - IPv4 (IPv4 协议)
    - vs. IPv6 (对比 Ipv6 协议), 435
  - number of addresses (地址数), 900, 904
  - IPv6 (Ipv6 协议)
    - vs. IPv4 (对比 IPv4 协议), 435
    - number of addresses (地址数), 901
  - IR (instruction register)(指令寄存器), 910
  - IR write control line (IR 写控制线), 1082
  - ISBN (International Standard BookNumber, 国际标准书号), 86
  - Isolated vertices in graphs (图中的孤立顶点), 703
  - Isomorphic binary trees (同构二叉树), 661
  - Isomorphism in graphs (图中的同构), 859
  - Items in collections (集合中的项), 566
  - Iterable interface (可迭代的接口), 451, 602
  - Iterable collections (可迭代集合), 601–605
    - arrays (数组), 603
    - linked lists (链表), 604–605
    - Queue (队列), 604–605
    - SET (集合), 652
    - Stack (栈), 603
  - Iterated function systems (迭代函数系统), 239–243
  - Iterations in BSTs (BST 中的迭代), 650
  - Iterator interface (迭代器接口), 451, 602–605
- ## J
- Java command (Java 命令), 3, 134
  - .java extension (.java 扩展名), 3, 6, 8, 197, 383
  - Java language (Java 语言)
    - benefits (好处), 9
    - libraries (库), 1094
    - overview (概述), 1–8
  - Java platform (Java 平台), 2
    - Java Virtual Machine (JVM, Java 虚拟机)
      - description (描述), 3
      - overview (概述), 965–966
      - as program (作为程序), 788
  - Java virtual machines (Java 虚拟机), 429
  - Johnson, David S. (大卫·约翰逊), 859
  - Josephus problem (约瑟夫问题), 619
  - Julia sets (朱利亚集合), 427

Jump and link instruction (跳转和链接指令), 931  
 Jump register instruction (跳转寄存器指令), 931

## K

K-ring graphs (K 阶环图), 694–695  
 K-way multiplexers (K 路多路复用器), 1019–1020  
   Kamasutra ciphers (Kamasutra 密码), 377  
   Karp, Richard (卡尔普·理查德), 845–848  
 Karp's reductions (卡尔普归约)  
   NP-completeness (NP 完全性), 845–848  
   polynomial-time (多项式时间), 841  
 Kevin Bacon game (凯文·贝肯游戏), 684–686  
 Key-sorted tree traversal (键有序树遍历), 649  
 Keys (键)  
   BSTs (二分搜索树), 640–642, 650  
   cryptographic (密码学), 992  
   immutable (不可变), 625  
   Kamasutra ciphers (Kamasutra 密码), 377  
   symbol tables (符号表), 624–626, 655  
 Key-value pairs (键-值对), 624–626  
 Kleene, Stephen (史蒂芬·克林), 748  
 Kleene's theorem (克林定理)  
   applications (应用), 753–756  
   DFA, NFA, and RE equivalence (DFA、NFA 和 RE 等价), 749–752  
   overview (概述), 748  
   power limitations (功率限制), 753–756  
   proof strategy (证明策略), 748  
   RE recognition (RE 识别), 753  
 Knuth, Donald (唐纳德·克努特)  
   MIX machine (MIX 机), 947  
   optimization (优化), 518  
   running time (运行时间), 496, 501  
   SAT solvers (SAT 解决器), 832  
 Koch program (程序 Koch), 397

## L

Ladders, word (词梯), 710  
 Ladner, R. (理查德·拉德纳), 859  
 Lambda calculus (Lambda 演算), 790, 794  
 Lambda expressions (Lambda 表达式), 450  
 Languages (语言)  
 Last-in first-out (LIFO, 后进先出), 566–567  
 Lattices in random walks (随机游走网格), 112–

115

Layers of abstraction (抽象层), 1037–1039  
 LCS (longest common subsequence, 最长公共子序列), 285–288  
 Leading zeros, 883  
 Leaf nodes in BSTs (二叉搜索树的叶子节点), 640  
 Leaks, memory (内存泄漏), 367, 581  
 LeapYear program (程序 LeapYear), 28–29  
 Left associativity (左结合), 17  
 Left shift operations (左移操作)  
   bitwise (按位), 891–892  
   TOY machine (TOY 机), 913  
 Left subtrees (左子树), 640  
 Length (长度)  
   arrays (数组), 91–92  
   graphs paths (图路径), 674, 683  
   strings (字符串), 332  
 Less than signs (<)  
   bitwise operations (按位运算), 891–892  
   comparisons (比较), 27–29  
   redirection (重定向), 140–141  
 Let's Make a Deal simulation (让我们进行交易模拟), 88  
 Levin, Leonid (莱昂纳德·莱维), 845  
 Liar's paradox (说谎者悖论), 807–808  
 Libraries (库)  
   APIs (应用程序编程接口), 230–232  
   array I/O (数组 I/O), 237–238  
   clients (客户端), 230  
   extensible (可扩展的), 452  
   Java, 1094  
   methods (方法), 29–32  
   modifying (修改), 255  
   in modular programming (模块化编程), 227–228, 251–254  
   modules (模块), 191  
   overview (概述), 226, 230  
   random numbers (随机数), 232–236  
   statistics (统计), 244–250  
   stress testing (压力测试), 236  
   unit testing (单元测试), 235  
 LIFO (last-in first-out, 后进先出), 566–567  
 Lights for TOY machine (TOY 机指示灯), 916  
 Lindenmayer systems (Lindenmayer 系统), 803

- Linear algebra for vectors (向量的线性代数), 442–443
  - Linear equation satisfiability problem (线性方程的可满足性问题), 830, 839
  - Linear feedback shift registers (LFSR, 线性反馈移位寄存器), 1000–1001
  - Linear inequality satisfiability problem (线性不等式可满足性问题), 831, 839
  - Linear interpolation (线性插值), 351
  - Linear order of growth (线性增长量级), 504–505, 507–508
  - Linear programming problem (线性编程问题), 831
  - Linearithmic order of growth (线性对数增长量级), 504–505, 507–508
  - Linked lists (链表)
    - circular (环路), 622
    - FIFO queues (FIFO 队列), 593, 596
    - hash tables (散列表), 636
    - iterable classes (可迭代类), 604–605
    - overview (概述), 571–574
    - stacks (栈), 574–576
    - summary (总结), 578
    - symbol tables (符号表), 635
    - traversal (遍历), 574, 577
  - Linked structures (链接结构)
  - LinkedStackOfStrings program (程序 LinkedStackOfStrings), 574–576
  - Links in BSTs (二叉搜索树中的链接), 640–642
  - Lipton, R. J. (理查德·利普顿), 856
  - Lissajous, Jules A. (朱勒·A. 李萨如), 168
  - Lissajous patterns (李萨如模式), 168
  - Lists, linked (链表)
  - Literals (常量)
    - array elements (数组元素), 116
    - binary and hex (二进制和十六进制), 891
    - booleans (布尔), 26
    - characters (字符), 18–19
    - description (描述), 15
    - floating-point numbers (浮点数), 24
    - integers (整数), 22
    - strings (字符串), 19, 334
  - Little's law (利特尔法则), 598
  - Load address instruction (加载地址指令), 1080
  - Load instructions (加载指令), 938, 1080
  - LoadBalance program (程序 LoadBalance), 606–607
  - Local clustering (局部聚类性), 693–694
  - Local variables (局部变量)
    - vs. instance variables (对比实例变量), 384
    - static methods (静态方法), 196
  - Logarithmic order of growth (对数增长量级), 503
  - Logarithmic spirals (对数螺旋), 398–399
  - Logical design (逻辑设计), 1008–1009
  - Logical instructions (逻辑指令), 912–913
  - Logical shifts (逻辑移位), 891–892
  - Logical switches (逻辑开关)
    - bus muxes (总线多路复用器), 1036
    - demultiplexers (多路分配器), 1022
    - multiplexers (多路复用器), 1020
  - Logo language (Logo 语言), 400
  - Loitering condition (游离情况), 581
  - Long data type (长数据类型), 24, 513
  - Long path problems (长路径问题), 829
  - Longest common subsequence (LCS, 最长公共子序列), 285–288
  - Longest path problem (最长路径问题), 838
  - LongestCommonSubsequence program (程序 LongestCommonSubsequence), 286–288
  - Lookup program (程序 Lookup), 630–632
  - Loops (循环)
  - Lost letter (丢失的信), 840
  - Lower bounds (下界), 826
  - Luminance (亮度), 343–345
  - Luminance program (程序 Luminance), 344–345
- ## M
- M/M/1 queues (M/M/1 队列), 597–600
  - MAC addresses (MAC 地址), 877
  - Machine-language programming (机器语言编程)
    - arrays (数组), 938–941
    - benefits (优点), 945
    - description (描述), 907
    - functions (函数), 931–933
    - overview (概述), 930
    - standard input (标准输入), 936–938
    - standard output (标准输出), 934–936
    - summary (总结), 945–946
    - TOY machine (TOY 机), 914

- Magnitude (模 / 极径 / 大小)
  - complex numbers (复数), 402–403
  - spatial vectors (空间向量), 442–443
- Magritte, René (勒内·马格里特), 363
- main() methods (main() 方法), 4–5
  - multiple (倍数), 229
  - transfer of control (转移控制), 193–194
- Majority function (表决器函数)
  - adder circuits (加法器电路), 1028–1030
  - sum-of-products circuits (积之和电路), 1027
  - truth tables for (真值表), 1025
- Mandelbrot, Benoît (本华·曼德布洛特), 297, 406
- Mandelbrot program (程序 Mandelbrot program), 406–409
- Maps (地图), Mercator projections (墨卡托投影), 48
- Markov, Andrey (安德烈·马尔可夫), 176
- Markov chains (马尔可夫链)
  - impact (影响), 184
  - mixing (混合), 179–184
  - overview (概述), 176
  - power method (乘幂方法), 180–181
  - squaring (自乘), 179–180
- Markov model paradigm (马尔可夫模型范例), 460
- Markov program (程序 Markov), 180–182
- Markov systems (马尔可夫系统), 802–803
- Markovian queues (马尔可夫队列), 597
- Marsaglia's method (马尔萨利亚方法), 85, 259
- Masking bitwise operations (屏蔽按位运算), 892–893
- Matcher class for REs (RE 的 Matcher 类), 763
- Matching graphs (匹配图), 713
- Math library (Math 库), 192
  - accessing (访问), 228
  - methods (方法), 30–32, 193, 198
- Mathematical analysis (数学分析), 498–502
- Mathematical functions (数学函数), 202–204
- Mathematical induction (数学归纳), 262, 266
- Mathematical models (数学模型), 716
- Matiyasevich, Yuri (尤里·马季亚谢维奇), 816
- Matlab language (Matlab 语言), 1094
- Matrices (矩阵)
  - boolean (布尔), 302
  - Hadamard (哈达玛), 122
  - images (图像), 346–347
  - matrix multiplication (矩阵乘法), 109
  - sparse (稀疏), 666
  - transition (转置), 172–173
  - two-dimensional arrays (二维数组), 106, 109–110
  - vector multiplication (向量乘法), 110, 180
- Mauchly, John (约翰·莫奇利), 924–925
- Maximum values in arrays (数组最大值), 209
- Maximum keys in BSTs (二叉搜索树的最大键), 651
- Maxwell-Boltzmann distributions (克斯韦-玻耳兹曼分布), 257
- McCarthy's 91 function (麦卡锡 91 函数), 298
- Mechanical systems, graphs for (机器系统图表), 673
- Memoization (记忆), 284
- Memory (内存)
  - arrays (数组), 91, 94, 515–517
  - ArrayStackOfStrings, 569–570
  - available (可用的), 520
  - bit-slice design (位片设计), 1054–1056
  - circuits (电路), 1054–1057
  - feedback loops as (反馈循环), 1048
  - flip-flops (触发器), 1049–1050
  - interfaces (接口), 1054
  - leaks (泄漏), 367, 581
  - linked lists (链表), 571
  - memory bits (内存位), 1056
  - objects (对象), 338, 514
  - performance (性能), 513–517
  - recursion (递归), 282
  - references (引用), 367
  - safe pointers (安全指针), 366
  - strings (字符串), 515
  - TOY machine (TOY 机), 908–909
  - two-dimensional arrays (二维数组), 107
  - virtual (虚拟), 972, 975–976
- Memory dumps (内存导出), 909
- Memory instructions (内存指令)
  - address instructions (地址指令), 912
  - TOY machine (TOY 机), 913
- Memory writes for CPU (CPU 内存写入), 1079
- Memoryless queues (无记忆队列), 597

- Mercator projections (墨卡托投影), 48
- Merge program (程序 Merge), 550–552
- Mergesort (归并排序)
  - divide-and-conquer (分而治之), 554
  - overview (概述), 550–552
  - performance (性能), 553
- Metacharacters (元字符), 724, 730–731
- Method references (方法引用), 470
- Methods (方法)
  - abstract (摘要), 446
  - call chaining (调用链), 404
  - deprecated (已弃用), 469
  - instance (实例), 334, 385–386
  - instance vs. static (实例与静态比较), 340
  - library (库), 29–32
  - main(), 4–5
  - overriding (重写), 452
  - static (静态)
  - stub (存根), 303
  - variables within (变量), 386–388
- MIDI Tuning Standard (MIDI 调制标准), 161
- Midpoint displacement method (中点偏移法), 278, 280
- Milgram, Stanley (斯坦利·米尔格拉姆), 670
- Minimum keys in BSTs (二叉搜索树中的最小键), 651
- Minsky, Marvin (马文·明斯基), 780, 794
- Minus signs (–)
  - compound assignments (组合赋值), 60
  - floating-point numbers (浮点数), 24–26
  - integers (整数), 22
  - lambda expressions (Lambda 表达式), 450
- MIX machine (MIX 机), 947
- Mixed-type operators (混合类型运算符), 27–29
- Mixing Markov chains (混合马尔可夫链), 176, 179–184
- MM1Queue program (程序 MM1Queue), 598–600
- Models (模型)
  - circuit (电路)
  - computational (计算), 716
  - mathematical (数学), 716
  - universal (通用), 794–797
- Modular programming (模块化编程), 191
  - classes in (类), 227–229
  - code reuse (代码复用), 226, 253
  - debugging (调试), 253
  - encapsulation (封装), 432
  - flow of control in (控制流程), 227–228
  - libraries in (库), 251–254
  - machine language (机器语言), 932
  - maintenance (维护), 253
  - program size (程序规模), 252–253
- Modules (模块)
  - abstraction layers (抽象层), 1037
  - as classes (作为类), 228
  - CPU (中央处理器), 1076
  - description (描述), 1034
  - interactions (交互), 319
  - overview (概述), 191
  - program counters (程序计数器), 1073
  - size (规模大小), 319
  - summary (总结), 254
- Monochrome luminance (单色亮度), 343–344
- Monte Carlo simulation (蒙特卡洛模拟), 300, 307–308
- Moore's Law (摩尔定律)
  - coping with (应对), 971
  - description (描述), 507–508
- Move-to-front strategy (“移动到前端”策略), 620
- Movie-performer graph (电影–演员图), 680
- Multidimensional arrays (多维数组), 111
- Multiple arguments (多个参数), 197
- Multiple inheritance (多个继承), 470
- Multiple main() methods (多个 main() 方法), 229
- Multiple return statements (多返回语句), 198
- Multiple I/O streams (多输入输出流), 143
- Multiplexers (多路复用器)
  - bus switching (总线切换), 1035
  - description (描述), 1023
  - selection (选择), 1019–1020
- Multiplication (乘法)
  - complex numbers (复数), 402–403
  - floating-point numbers (浮点数), 24–26
  - integers (整数), 22–23, 885
  - matrices (矩阵), 109–110
  - P search problems (P 搜索问题), 839
  - polar representation (极坐标表示), 433

Multiway gates (多路门), 1015–1017, 1023

Music (音乐), 155–159

Mutable types (可变类型), 364, 439

## N

N-body simulation (多体模拟)

Body data type (天体数据类型), 479–480

file format (文件格式), 483

force (力), 480–482

overview (概述), 478–479

summary (总结), 488

Universe data type (Universe 数据类型), 483–487

Names (名称)

arrays (数组), 91

methods (方法), 5, 30, 196

objects (对象), 362

variables (变量), 16

vertices (顶点), 675

NaN value (NaN, 非数值), 26, 40

NAND function (NAND 函数), 989–991

Nash, John (约翰·纳什), 840

Natural numbers (自然数), 875

Natural recursion (自然递归), 262

Negation axiom (否定公理), 990

Negative numbers (负数)

array indexes (数组索引), 116

representing (表示), 38, 886–888

Neighbor vertices (邻居顶点), 671

Nested classes (嵌套类)

iterators (迭代器), 574

linked lists (链表), 603–605

Nesting conditionals and loops (嵌套条件和循环), 62–64

new keyword (new 关键字)

constructors (构造函数), 385

Node objects (Node 对象), 609

String objects (String 对象), 333

Newcomb, Simon (西蒙·纽科姆), 224

Newline characters (\\n, 新行)

compiler considerations (编译环境), 10

escape sequences (转义序列), 19

Newton, Isaac (艾萨克·牛顿)

dice question (骰子问题), 88

motion simulation (运动模拟), 478–479

square root method (平方根计算), 65

Newton's law of gravitation (牛顿万有引力定律), 481

Newton's method (牛顿迭代法), 65–67

Newton's second law of motion (牛顿第二运动定律), 480–481

NFAs (非确定有限状态自动机)

90–10 rule (90-10 规则), 170, 176

Nodes (节点)

BSTs (二分搜索树), 640–642, 942

linked lists (链表), 571–573

new keyword (new 关键字), 609

Nondeterministic finite-state automata (NFA, 非确定有限状态自动机)

DFA equivalence (DFA 等价), 749–750

Kleene's theorem (克林定理)

overview (概述), 744

RE equivalence (RE 等价), 750–751

recognition problem (识别问题), 744–745

trace example (追踪示例), 747

Nondominant inner loop (非主导地位的内循环)s, 510

NOR function (NOR 函数), 989–991

NOR gates (或非门)

cross-coupled (交叉耦合), 1050

description (描述), 1014

Normal distribution functions (正态分布函数)

cumulative (累计), 202–203

probability density (概率密度), 202–203

NOT gates (非门), 1013–1014

Not operation (非操作), 26–27, 987–989

NP-completeness (NP 完全性)

addressing problems (解决问题), 852

boolean satisfiability (布尔可满足性), 853–856

classifying problems (分类问题), 851

Cook-Levin theorem (库克-莱文定理), 844–847

coping (应对), 850–857

Karp's reductions (卡尔普归约), 845–848

overview (概述), 843–844

proving (证明), 844–849

NP-hard problems (NP 难问题), 858

NP search problems (NP 查找问题)

difficult (困难的), 837

easy (简单的), 837

main question (主要问题), 840–841  
 nondeterminism (非确定性), 835  
 overview (概述), 833  
 solutions (解决方法), 835  
 subset sum (子集和), 834  
 TSP problem (TSP 问题), 862  
 vertex cover problem (顶点覆盖问题), 834, 842  
 0/1 ILP problem (0/1 ILP 问题), 835  
 Null calls (空调用), 312  
 Null keys in symbol tables (符号表中的空键), 626  
 Null links in BSTs (二叉搜索树中的空链接), 640  
 Null nodes in linked lists (二叉搜索树中的空节点), 571–572  
 null keyword (null 关键字), 415  
 Null transitions in NFAs (NFA 中的空转换), 744–746  
 Null values in symbol tables (符号表中的空值), 626  
 NullPointerException, 370  
 Numbers (数字)  
   conversions (转换), 21, 67–69, 880–881  
   intractability (难解性), 827  
   negative (负的), 886–888  
   real (真值), 888–890  
 Numerical integration (数值积分), 449  
 Nyquist frequency (奈奎斯特频率), 161

## O

Object class (对象类), 453–455  
 Object-oriented programming (面向对象编程)  
   data types (数据类型)  
   description (描述), 254  
   overview (概述), 329  
 Objects (对象)  
   arrays (数组), 365  
   collections (集合), 582–583  
   comparing (比较), 364, 545–546  
   Complex (复数), 404  
   equality (相等性), 454–456  
   memory (内存), 514  
   names (名称), 362  
   orphaned (孤立), 366  
   references (引用), 338–339

String, 333–334  
 type conversions (类型转换), 339  
 uninitialized variables (未初始化的变量), 339  
 working with (共同协作), 338–339  
 Observations (观察), 495–496  
 Occam's Razor (奥卡姆的剃刀), 814  
 Octal representation (八进制表示), 898  
 Odd parity function (奇偶校验函数)  
   adder circuits (加法器电路), 1028–1030  
   sum-of-products circuits (积之和电路), 1026  
   truth tables for (真值表), 1026  
 Off-by-one errors (数据访问错位), 92  
 Offscreen canvas (缓冲区画布), 151  
 Offset binary representation (偏移二进制表示), 889  
 On computable numbers, with an application to the Entscheidungsproblem article (文章《可计算数及其在决策问题上的应用》), 717  
 On/off switches (打开/关闭开关), 1005  
 One-dimensional arrays (一维数组), 90  
 One-hot OR gates (单热或门), 1023  
 Onscreen canvas (屏幕画布), 151  
 Opcodes (操作码), 911  
 Operands (操作数), 17  
 Operators and operations (操作符和运算操作)  
   boolean (布尔), 26–27, 989–991  
   comparisons (比较), 27–29, 364  
   compound assignments (组合赋值), 60  
   data types (数据类型), 14, 331  
   description (描述), 15  
   expressions (表达式), 17, 587  
   floating-point numbers (浮点数), 24  
   integers (整数), 22, 891  
   lambda (Lambda), 450  
   overloading (重载), 416  
   precedence (优先级), 17  
   reverse Polish notation (反向波兰表示法), 590  
   stacks (栈), 590  
   strings (字符串), 19, 21, 334, 453  
   TOY machine (TOY 机), 906  
 Optimal data compression (最佳数据压缩), 814  
 Optimization (优化)  
   NP problems (NP 问题), 835  
   premature (不成熟的), 518



Optimizing compilers (优化编译器), 814  
 OR function (OR 函数), 987–989  
 OR gates (或门), 1014, 1023  
 Or operation (or 运算符)  
   bitwise (按位), 891–892  
   boolean type (布尔类型), 26–27  
   TOY machine (TOY 机), 913  
 Order in BSTs (二叉搜索树中的有序操作), 640, 642–643  
 Order statistics (层序统计), 651  
 Order-of-growth classifications (增长量级分类)  
   constant (常数), 503  
   cubic (立方), 505–508  
   exponential (指数), 505–508  
   linear (线性), 504–505, 507–508  
   linearithmic (线性对数), 504–505, 507–508  
   logarithmic (对数), 503  
   overview (概述), 503  
   performance analysis (性能分析), 500–501  
   quadratic (二次), 504–505, 507–508  
 Ordered operations (有序操作)  
   binary search trees (二叉搜索树), 651  
   symbol tables (符号表), 624  
 Orphaned objects (孤立对象), 366  
 Orphaned stack items (孤立的堆栈项), 581  
 Out library (出库), 355–356  
 Outer loops (外循环), 62  
 Outline shapes (图形的描边), 149  
 Output (输出)  
   arithmetic logic units (算术逻辑单元), 1032  
   array libraries (数组库), 237–238  
   circuit models (电路模型), 1002–1004  
   clocks (时钟), 1059–1060  
   data types (数据类型), 353  
   file concatenation (文件连接), 356  
   gates (门), 1013  
   machine language (机器语言), 934–936  
   print statements (打印语句), 8  
   printf() method (printf() 方法), 126–129  
   standard (标准), 127, 129–132  
   standard audio (标准音频), 155–159  
   standard drawing (标准绘图)  
   stream data types (流式数据类型), 355  
   two-dimensional arrays (二维数组), 107

virtual machines (虚拟机), 969–970  
 Overflow (溢出)  
   arithmetic (算术), 885  
   arrays (数组), 95  
   attacks (攻击), 963  
   guarding against (防御), 898  
   integers (整数), 23  
   negative numbers (负数), 38  
 Overhead for objects (对象的开销), 514  
 Overloading (超载)  
   operators (运算符), 416  
   static methods (静态方法), 198  
 Overriding methods (重写方法), 452

## P

The P= NP Question and Gödel's Lost Letter book  
   (P = NP 问题和哥尔德丢失的信件), 856  
 P search problems (P 搜索问题), 837  
   examples (示例), 839  
   main question (主问题), 840–841  
 Padding object memory (填充对象存储器), 514  
 Page, Lawrence (劳伦斯·佩奇), 184  
 Page ranks (页面排名), 176–177  
 Palindromes (回文)  
   description (描述), 719  
   Watson-Crick (沃森-克里克), 374  
 Paper size (纸张大小), 294  
 Paper tape (纸带), 934–938  
 Papert, Seymour (西蒙·派珀特), 400  
 Parallel arrays (平行数组), 411  
 Parallel edges (平行边), 676  
 Parameter variables (参数变量)  
   lambda expressions (Lambda 表达式), 450  
   static methods (静态方法), 196–197, 207  
 Parameterized data types (参数化数据类型), 582–586  
 Parameters (参数)  
   in performance (性能), 511  
   TOY-8 machine (TOY-8 计算机), 1070  
 Parentheses ()  
   casts (转型), 33  
   constructors (构造函数), 333, 385  
   expressions (表达式), 17, 27  
   functions (函数), 24, 197  
   lambda expressions (Lambda 表达式), 450

- methods (方法), 30, 196
- operator precedence (运算符优先级), 17
- regular expressions (正则表达式), 724
- stacks (栈), 587, 590
- static methods (静态方法), 196
- vectors (向量), 442
- Parity in ripple-carry adders (进位加法器中的奇偶校验), 1028
- Parsing (解析)
  - instructions (指令), 966–967
  - strings (字符串), 880–882
- Pascal's triangle (帕斯卡三角形), 125
- Passing arguments (传递参数)
  - references by value (值引用), 364–365
  - static methods (静态方法), 207–210
- PathFinder program (程序 PathFinder), 683–686, 690–692
- Paths (路径)
  - graphs (图), 674, 683–692
  - intractability problems (难以解决的问题), 829
  - shortest (最短)
  - simple (简单), 710
- Pattern class for REs (RE 的 Pattern 类), 763
- PCs
- PDA (pushdown automata)(下推自动机), 755–756
- PDP-8 computers (PDP-8 计算机), 906
- Peaks in terrain analysis (地形分析中的峰值), 167
- Pell's equation (佩尔方程), 869
- Pens (笔)
  - color (颜色), 150
  - drawings (绘图), 146
- Pepys, Samuel (塞缪尔·佩皮斯), 88
- Pepys problem (佩皮斯问题), 88
- Percent signs (%)(百分号)
  - conversion codes (转换代码), 131–132
  - remainder operation (求余运算符), 22–23
- Percolation case study (渗透案例研究)
  - adaptive plots (自适应绘图), 314–318
  - lessons (经验总结), 318–320
  - overview (概述), 300–301
- Percolation, 303–304
- PercolationPlot, 315–317
- PercolationProbability, 310–311
- PercolationVisualizer, 308–309
- probability estimates (概率估计), 310–311
- recursive solution (递归解决方案), 312–314
- scaffolding (脚手架), 302–304
- testing (测试), 305–308
- vertical percolation (垂直渗透原理), 305–306
- Performance (性能)
  - binary search trees (二叉搜索树), 647–648
  - binary searches (二分查找), 535
  - caveats (陷阱), 509–511
  - comparing (比较), 508–509
  - guarantees (保障), 512, 627
  - hypotheses (假设), 496–502
  - importance (重要性), 702
  - insertion sorts (插入排序), 544–545
  - memory use (内存使用), 513–517
  - mergesort (归并排序), 553
  - multiple parameters (多个参数), 511
  - order of growth (增长量级), 503–506
  - overview (概述), 494–495
  - perspective (展望), 518
  - prediction (预测), 507–509
  - scientific method (科学方法), 495–502
  - shortest paths (最短路径), 690
  - wrapper types (封装类型), 369
- Performer program (程序 Performer), 697–699
- Periods (.)
  - classes (类), 227
  - regular expressions (正则表达式), 724
- Permutations (排列)
  - inverse (反向), 122
  - sampling (抽样), 97–99
- Phase transitions (相变), 317
- Phone books (电话簿), 628
- Photographs (照片), 346
- Physical systems, graphs for (物理系统图表), 672
- Pi constant (pi 常量), 31–32
- Picture library (Picture 库), 346–347
- Piecewise approximation (分段近似), 148
- Pigeonhole principle (鸽笼原理), 754–755
- Piping (管道)
  - connecting programs (连接程序), 141
  - filters (过滤器), 142–143
- Pixels in image processing (图像处理中的像素), 346

- Plasma clouds (等离子体云), 280
- Playing card possibilities (纸牌游戏可能性), 823
- PlayThatTune program (程序 PlayThatTune), 157–158
- PlayThatTuneDeluxe program (程序 PlayThatTune-Deluxe), 213–215
- PlotFilter program (程序 PlotFilter), 146–147
- Plotting (绘图)
  - array values (数组值), 246–248
  - experimental results (实验结果), 249–250
  - function graphs (函数图形), 148, 248
  - percolation case study (渗透案例研究), 314–318
  - sound waves (声波), 249
- Plus signs (+)
  - compound assignments (组合赋值), 60
  - floating-point numbers (浮点数), 24–26
  - integers (整数), 22
  - regular expressions (正则表达式), 731
  - string concatenation (字符串拼接), 19–20
- Pointers (指针)
  - array elements (数组元素), 94
  - handles (句柄), 371
  - object references (对象引用), 338
  - safe (安全), 366
- Poisson processes (泊松过程), 597
- Polar coordinates (极坐标), 47
- Polar representation (极坐标表示), 433–434
- Polling, statistical (统计投票), 167
- Polymorphism (多态性), 448
- Polynomial time (多项式时间), 823
- Polynomial-time algorithms (多项式时间算法)
  - intractability (难解性), 825–826
  - P search problems (P 搜索问题), 837, 839
  - usefulness (有用), 858
- Polynomial-time reductions (多项式时间归约), 841–843
- Pop operation (出栈操作)
  - reverse Polish notation (反向波兰表示法), 590–591
  - in stacks (栈中), 567–568
- Positional notation (位置表示), 875
- Post, Emil (邮件), 813–814
- Post correspondence problem (邮寄通信问题), 813–814
- Postconditions in assertions (断言中的后置条件), 467
- Postfix notation (后缀表示法), 590
- Postorder tree traversal (后序树遍历), 649
- PostScript language (PostScript 语言), 400, 590
- PotentialGene program (程序 PotentialGene), 336–337
- Pound signs (#), 769
- Power method (乘幂方法), 180–181
- Power source (电源), 1003–1004
- PowersOfTwo program (程序 PowersOfTwo), 56–58
- Precedence (优先级)
  - arithmetic operators (算术运算符), 17
  - regular expressions (正则表达式), 724
- Precision (精度)
  - floating-point numbers (浮点数), 25, 40
  - printf(), 130–131
  - standard output (标准输出), 129–130
- Precomputed array values (预计算的数组值), 99–100
- Preconditions in assertions (断言的前置条件), 467
- Prediction, performance (性能预测), 507–509
- Preferred attachment process (优先连接过程), 713
- Prefix-free strings (无前缀字符串), 564
- Premature optimization (不成熟的优化), 518
- Preorder tree traversal (前序树遍历), 649
- Primality-testing function (原始测试函数), 198–199
- Prime numbers (质数)
  - in factoring (因数), 72–73
  - Sieve of Eratosthenes (埃拉托斯特尼筛法), 103–105
- PrimeSieve program (程序 PrimeSieve), 103–105
- Primitive data types (基本数据类型), 14
  - memory size (内存大小), 513
  - overflow checking (溢出检查), 39
  - performance (性能), 369
  - wrappers (封装), 457
- Principle of superposition (叠加原理), 483
- print() method (print() 方法), 31
  - arrays (数组), 237–238
  - impurity (杂项), 32

- Out, 355
  - vs. println() (对比 println()), 8
  - standard output (标准输出), 129–130
  - Print statements (打印语句), 5
  - printf() method (printf() 方法), 129–132, 355
  - Printing, formatted (格式化打印), 130–132
  - println() method (println() 方法), 31
    - description (描述), 5
    - impurity (杂项), 32
  - Out, 355
  - vs. print() (对比 print()), 8
  - standard output (标准输出), 129–130
  - string concatenation (字符串拼接), 20
  - private keyword (private 关键字)
    - access modifier (访问修饰符), 384
    - encapsulation (封装), 433
  - Probabilities (概率), 308, 310–311
  - Probability density function (概率密度函数), 202–203
  - Problem reduction (问题归约)
    - overview (概述), 811
    - program equivalence (等价程序), 812
    - Rice's theorem (莱斯定理), 812–813
    - totality problem (总和问题), 811–812
  - Problem size in intractability (问题规模大小的难处理性), 824
  - Procedural programming style (程序编程风格), 329
  - Program counters (PC, 程序计数器)
    - bus connections (总线连接), 1073–1074
    - connections and timing (连接与计时), 1075
    - control lines (控制线), 1074–1075
    - interfaces (接口), 1073
    - modules (模块), 1073
    - overview (概述), 1073
    - TOY machine (TOY 机), 910
  - Program equivalence problem (程序等价问题), 812
  - Program size (程序大小), 252–253
  - Programming environments (编程环境), 1094
  - Programming languages (编程语言)
    - indexing (索引), 634
    - stack-based (基于栈), 590
    - symbol tables (符号表), 629
  - Programming overview (编程概述), 1
    - HelloWorld example (HelloWorld 示例), 4–6
    - input and output (输入和输出), 7–8
    - process (处理), 2–3
  - Programs (程序)
    - connecting (连接), 141
    - processing programs (处理程序), 788–790, 964–966
  - Proof by contradiction (矛盾证明), 754
  - Pseudo-code (伪代码), 911
  - public keyword (public 关键字)
    - access modifiers (访问修饰符), 384
    - description (描述), 228
    - static methods (静态方法), 196
  - Pulses, clock (脉冲时钟), 1058
  - Punched cards (打孔卡), 940
  - Punched paper tape (打孔纸带), 934–938
  - Pure functions (纯函数), 201
  - Pure methods (纯方法), 32
  - Push operation (入栈操作)
    - reverse Polish notation (反向波兰表示法), 590–591
    - stacks (栈), 567–568
  - Pushbuttons for TOY machine (TOY 机的按钮), 916
  - Pushdown automata (下推自动机), 755–756
  - Pushdown stacks (下推栈), 567–568
  - Put operations (Put 操作)
    - hash tables (散列表), 639
    - symbol tables (符号表), 624
  - Putnam, Hilary (希拉里·普特南), 816
- ## Q
- Quad play (《Quad》剧本), 273
  - QuadraticKoch island fractal (方形科赫曲线), 803
  - Quadratic order of growth (二次型增长量级), 504–505, 507–508
  - Quadratic program (程序 Quadratic), 25–26
  - Quadrature integration (正交积分), 449
  - Quaternions (四元数), 424
  - Question marks (?) in REs (正则表达式中的问号), 731
  - Questions program (程序 Questions), 533–535
  - Queue program (程序 Queue), 592–596, 604–605
  - Queues (队列)
    - circular (环形), 620

deques (双端队列), 618  
 FIFO (先进先出)  
 overview (概述), 566  
 random (随机), 596  
 summary (总结), 608  
 Queuing theory (排队论), 597–600  
 Quotes (") in text (文本中的引号), 5

## R

Race conditions in flip-flops (触发器竞争条件), 1050  
 Ragged arrays (交错数组), 111  
 Ramanujan, Srinivasa (斯里尼瓦瑟·拉马努金), 86  
 Ramanujan's taxi (拉马努金的出租车问题), 86  
 Random graphs (随机图), 695  
 Random numbers (随机数)  
   fair coin flips (公平硬币翻转), 52–53  
   function implementation (函数实现), 199  
   Gaussian (高斯), 47  
   impurity (杂质), 32  
   libraries (库), 232–236  
   random sequences (随机序列), 127–128  
   Sierpinski triangles (谢尔宾斯基三角形), 239–240  
   simulations (模拟), 72–73  
   Math.random(), 30–31  
 Random queues (随机队列), 596  
 Random shortcuts (随机捷径), 699  
 Random walks (随机游走)  
   Brownian bridges (布朗桥), 278  
   self-avoiding (自回避), 112–115  
   two-dimensional (二维), 86  
   undirected graphs (无向图), 712  
 Random web surfer case study (随机网络冲浪案例研究)  
   histograms (直方图), 177  
   input format (输入格式), 171  
   lessons (经验总结), 184–185  
   Markov chains (马尔可夫链), 176, 179–184  
   overview (概述), 170–171  
   page ranks (页面排名), 176–177  
   simulation (模拟), 174–178  
   transition matrices (转置矩阵), 172–173  
 RandomInt program (程序 RandomInt), 33–34

RandomSeq program (程序 RandomSeq), 127–128  
 RandomSurfer program (程序 RandomSurfer), 175–177  
 RangeFilter program (程序 RangeFilter), 140–143  
 Ranges (范围)  
   binary search trees (二叉搜索树), 651  
   functions (函数), 192  
 Ranks (排名)  
   binary search trees (二叉搜索树), 651  
   random web surfer (随机网上冲浪), 176–177  
 Raphson, Joseph (约瑟夫·拉弗森), 65  
 Raster images (光栅图像), 346  
 Real numbers (实数), 888–890  
 Receivers in cryptography (密码学中的接收者), 992  
 Recognition problem (识别问题)  
   formal languages (形式语言), 722  
   NFAs (非确定有限状态自动机), 744–745  
   REs (正则表达式), 728–729, 735, 753  
 Recomputation (验算), 282–283  
 Rectangle rule (矩形法则), 449  
 Recurrence relations (递归关系), 272  
 Recursion (递归), 191  
   base cases (基础步骤), 281  
   BSTs (二叉搜索树), 640–641, 644, 649  
   binary searches (二分查找法), 533  
   Brownian bridges (布朗桥), 278–280  
   considering (考虑), 320  
   convergence issues (收敛问题), 281–282  
   dynamic programming (动态编程), 284–289  
   Euclid's algorithm (欧几里得算法), 267–268  
   exponential time (指数时间), 272–273  
   factorial example (阶乘例子), 264–265  
   function-call trees (函数调用树), 269, 271  
   graphics (图形), 276–277, 397  
   Gray codes (格雷码), 273–275  
   linked lists (链表), 571  
   mathematical induction (数学归纳法), 266  
   memory requirements (内存要求), 282  
   mergesort (归并排序), 550  
   overview (概述), 262–263  
   percolation case study (渗透案例研究), 312–314  
   perspective (展望), 289  
   pitfalls (陷阱), 281–283

- recomputation issues (验算问题), 282–283
- towers of Hanoi (汉诺塔), 268–272
- Red-black trees (红黑树), 648
- Redirection (重定向), 139
  - pipng (管道), 142–143
  - standard input (标准输入), 140–141
  - standard output (标准输出), 139–140
- Reduced instruction set computing (RISC) (精简指令集计算), 974
- Reductio ab absurdum (反证法), 808
- Reduction (归约)
  - binary search trees (二叉搜索树), 640
  - mergesort (归并排序), 554
  - polynomial-time (多项式时间), 841–843
  - problem (问题), 811–813
  - recursion (递归), 264–265
- References (引用)
  - accessing (访问), 339
  - aliasing (别名), 363
  - arrays (数组), 365
  - equality (相等性), 454–455
  - garbage collection (垃圾回收), 367
  - immutable types (不可变类型), 364, 441
  - linked lists (链表), 572
  - memory (内存), 367
  - method (方法), 470
  - object-oriented programming (面向对象编程), 330
  - objects (对象), 338–339
  - orphaned objects (孤立对象), 366
  - passing (传递), 207, 210, 364–365
  - performance (性能), 369
  - properties (属性), 362–363
  - safe pointers (安全指针), 366
- Reflexive property (自反性属性), 454
- Registers (寄存器)
  - implementing (实现), 1052
  - machine language (机器语言), 931
  - overview (概述), 1051–1052
  - TOY machine (TOY 机), 909, 911
  - writing to (写入), 1052–1053
- Regular expressions (RE, 正则表达式)
  - applications (应用), 732–736
  - computational biology (计算生物学), 732–734
  - generalized (广义), 730–732
  - NFA equivalence (NFA 等价), 750–752
  - overview (概述), 724–725
  - recognition problem (识别问题), 728–729, 735, 753
  - regular languages (正则语言), 725–727
  - searches (搜索), 734–736
  - shorthand notations (简写符号), 730–731
  - validity checking (有效性检查), 732
- Regular languages (正则语言), 723
  - basic operations (基本操作), 723–724
  - regular expressions (正则表达式)
- Reject states (拒绝状态)
  - DFA (确定有限状态自动机), 738–739
  - Turing machines (图灵机), 766–767
- Relative entropy (相对熵), 667–668
- Relays in circuit models (电路模型中的继电器), 1006
- Remainder operation (取余运算操作), 22–23
- Removing (移除)
  - array items (数组项), 569
  - collection items (集合项), 566, 602–603
  - linked list items (链表项), 573–574
  - NFA nodes (NFA 节点), 751
  - queue items (队列项), 592, 596
  - set keys (集合键), 652
  - stack items (栈项), 567–569
  - symbol table keys (符号表键), 624–627
- Rendell, Paul (保罗·兰德尔), 805
- Repetitive code, simplifying (简化的重复代码), 100
- Representation in APIs (应用程序编程接口中的表示), 431
- Representing information (表示信息)
  - binary and hex (二进制和十六进制), 875–880
  - bit manipulation (位操作), 891–893
  - characters (字符), 894–895
  - integer arithmetic (整数运算), 884–885
  - negative numbers (负数), 886–888
  - overview (概述), 874
  - real numbers (实数), 888–890
  - strings (字符串), 880–883
  - summary (总结), 896
- Reproducible experiments (可重复的实验), 495

Reserved words (保留字), 16

Resetting flip-flops (重置触发器), 1050

Resizing arrays (可变数组), 578–581, 635

ResizingArrayStackOfStrings program (程序 Resizing-ArrayStackOfStrings), 578–581

Resource allocation (资源分配)

- graphs for (图表), 673
- overview (概述), 606–607

Resource-sharing systems (资源共享系统), 606–607

Return addresses (返回地址), 931

return statements (返回语句), 194, 196, 198

Return values (返回值)

- arrays as (数组), 210
- methods (方法), 30, 196, 200, 207–210
- reverse Polish notation (反向波兰表示法), 591

Reuse, code (代码复用), 226, 253, 701

Reverse Polish notation (反向波兰表示法), 590

RGB color format (RGB 颜色格式), 48–49, 341, 371

Rice, Henry (亨利·莱斯), 812

Rice's theorem (莱斯定理), 812–813

Riemann integral (黎曼积分), 449

Riffle shuffles (鸽尾式洗牌), 125

Right shift operations (右移操作)

- bitwise (按位), 891–892
- TOY machine (TOY 机), 913

Right subtrees (右子树), 640

Right triangles (直角三角形), 199

Ring buffers (环形缓冲器), 620

Ring graphs (环形图), 694–695, 699

Ripple-carry adders (进位加法器), 1028–1030

RISC (reduced instruction set computing) (精简指令集计算), 974

Robinson, Julia (茱莉亚·罗宾逊), 816

Roots in binary search trees (二叉搜索树的根), 640

Rotation filters (旋转过滤器), 379

Roulette-wheel selection (轮询选择), 174

Round-robin policies (轮询调度策略), 606

Rows in 2D arrays (二维数组中的行), 106, 108

RR-format instructions (RR 指令格式), 911

Ruler program (程序 Ruler), 19–20

Run-time errors (运行时错误), 6

Running time (运行时间)

Running virtual machines (运行虚拟机), 969

RuntimeException, 466

## S

Safe pointers (安全指针), 366

Sample program (示例程序), 98–99

Sample standard deviation (样本标准差), 246

Sample variance (样本方差), 244

Sampling (采样)

- audio (音频), 156–157
- function graphs (函数图), 148
- scaling (缩放), 349–350
- without replacement (无放回), 97–99

SAT problem (SAT 问题), 832

- nondeterministic TM (非确定性 TM), 836
- NP-completeness (NP 完全性), 844–846, 853–856

SAT program (程序 SAT), 855–856

Satisfiability (可满足性), 830

- boolean (布尔), 832, 836
- integer linear inequality (整数线性不等式), 831
- linear equation (线性方程), 830
- linear inequality (线性不等式), 831
- NP-completeness (NP 完全性), 844–846, 853–856

Saving audio files (保存音频文件), 157

Scaffolding (脚手架), 302–304

Scale program (程序 Scale), 349–350

Scaling (缩放)

- drawings (绘图), 146
- image processing (图像处理), 349–350
- spatial vectors (空间向量), 442–443

Scientific computing (科学计算), 1094

Scientific method (科学方法), 494–495

- hypotheses (假说), 496–502
- observations (观察), 495–496

Scientific notation (科学计数法)

- conversion codes (转换代码), 131–132
- real numbers (实数), 888–889

Scope of variables (变量的范围), 60, 200

Screen scraping (屏幕抓取), 357–359

Search problems (搜索问题)

- difficult (困难的), 837–838
- easy (简单的), 837



- nondeterminism (非确定性), 836
- overview (概述), 833
- solutions (解决方案), 835
- subset sum (子集和), 834
- TSP problem (TSP 问题), 862
- vertex cover problem (顶点覆盖问题), 834, 842
- 0/1 ILP problem (0/1 ILP 问题), 835, 842
- Searches (搜索)
  - binary (二分)
  - binary search trees (二叉搜索树)
  - bisection (按位), 537
  - breadth-first (广度优先), 683, 687–688, 690, 692
  - data mining example (数据挖掘示例), 458–464
  - depth-first (深度优先), 312, 690
  - indexing (索引), 634
  - overview (概述), 532
  - regular expressions (正则表达式), 734–736
  - for similar documents (相似文件), 464
- Secret messages (加密消息), 992
- Seeds for random numbers (随机数的种子), 475
- Select control lines (选择控制线), 1056
- Self-avoiding walks (自回避行走), 112–115, 710
- Self-loops for edges (边缘的闭环), 676
- Self-modifying code (自修改代码), 922–924
- SelfAvoidingWalk program (程序 SelfAvoidingWalk), 112–115
- Semantics (语义学), 52
- Semicolons (;)
  - for loops (for 循环), 59
  - statements (语句), 5
- Sequential circuits (时序电路)
  - clocks (时钟), 1058–1061
  - description (描述), 1008
  - feedback circuits (回馈电路), 1048–1049
  - flip-flops (触发器), 1049–1050
  - memory (内存), 1054–1057
  - overview (概述), 1048
  - registers (寄存器), 1051–1053
  - summary (总结), 1062–1063
- Sequential searches (顺序查找), 535–536
- Server farms (服务器农场), 976
- Servers (服务器), 606
- Service rate (服务率), 597–598
- SET library (SET 库), 652–653
- Sets (集合)
  - elementary functions (基本功能), 1001
  - gates (门), 1045
  - graphs (图), 676
  - Julia (朱莉娅), 427
  - Mandelbrot (曼德布洛特集合), 406–409
  - overview (概述), 652–653
  - of values (值), 14
- Setting flip-flops (设置触发器), 1050
- Shadow variables (映射实例变量), 419
- Shannon, Claude (克劳德·香农), 1013, 1041
- Shannon entropy (香农熵), 378
- Shapes, outline and filled (形状、轮廓和填充), 149
- Shifts (移位)
  - bits (位), 891–892
  - circular (环形), 375
  - linear feedback shift registers (LFSRs) (线性反馈移位寄存器), 1000–1001
  - purpose (目标), 898–899
  - TOY machine (TOY 机), 913
- short data type (short 数据类型), 24
- Shortcuts in ring graphs (环形图中的捷径边), 699
- Shortest paths (最短路径)
  - adjacency-matrix (邻接矩阵), 692
  - breadth-first searches (广度优先搜索), 690
  - degrees of separation (分离度), 684–686
  - distances (距离), 687–688
  - graphs (图), 674, 683
  - implementation (实施), 691
  - P search problems (P 搜索问题), 829, 839
  - performance (性能), 690
  - single-source clients (单源客户端), 684
  - trees (树), 688–689
- Shuffling arrays (混排数组), 97
- Sicherman dice (赛克文骰子), 259
- Side effects (副作用)
  - arrays (数组), 208–210
  - assertions (断言), 467
  - importance (重要性), 217
  - methods (方法), 32, 126, 201
- Sierpinski triangles (谢尔宾斯基三角形), 239–

- 240
- Sieve of Eratosthenes (埃拉托斯特尼筛法), 103–105
- Sign-and-magnitude (符号加数值表示法), 886
- Sign extension convention (符号扩展转换), 899
- Signatures (签名)
- constructors (构造函数), 385
  - methods (方法), 30, 196
  - overloading (超载), 198
- Similarity measures (相似性度量), 462
- Simple paths (简单路径), 710
- Simplex method (单纯形方法), 831
- Simulations (模拟)
- coupon collector (卡券收集器), 174–178
  - dice (骰子), 121
  - gambler's ruin (赌徒破产), 69–71
  - Let's Make a Deal, 88–89
  - load balancing (负载均衡), 606–607
  - M/M/1 queues (M/M/1 队列), 598–600
  - Monte Carlo (蒙特卡洛), 300, 307–308
  - n-body (多体)
  - random web surfer (随机网络冲浪), 174–178
- Single-line comments (单行注释), 5
- Singles quotes (')(单引号), 19
- Singly linked lists (单链表), 571
- Sipser, Michael (迈克尔·西普塞), 780
- Six degrees of separation (六度分隔), 670
- Size (规模大小)
- arrays (数组), 578–581, 635
  - binary search trees (二叉搜索树), 651
  - modules (模块), 319
  - paper (纸带), 294
  - problems (问题), 495, 824
  - program (程序), 252–253
  - symbol tables (符号表), 624
  - words (单词), 874, 897
- Sketch program (程序 Sketch), 459–462
- Sketches (文档摘要)
- comparing (比较), 462–463
  - computing (计算), 459–460
  - hashing (散列), 460
  - overview (概述), 458–459
- Slashes (/)
- comments (注释), 5
  - floating-point numbers (浮点数), 24–26
  - integers (整数), 22–23
- Slide rules (算尺), 907–908
- Small-world case study (小世界案例研究)
- Small-world phenomenon (小世界现象), 670, 693
- SmallWorld program (程序 SmallWorld), 696
- Smith–Waterman algorithm(史密斯·沃特曼算法), 286
- Social network graphs (社交网络图), 672
- Sorts (排序)
- Arrays.sort(), 559
  - frequency counts (频率计数器), 555–557
  - insertion (插入), 543–549
  - lessons (经验总结), 558
  - mergesort (归并排序), 550–555
  - overview (概述), 532
  - P search problems (P 搜索问题), 839
- Sound (声音)
- Sound waves (声波)
- plotting (绘图), 249
  - superposition of (叠加), 211–215
- Source vertices (源顶点), 683
- Space-filling curves (空间填充曲线), 425
- Spaces (空间), 10
- Space–time tradeoff (时间和空间的折中), 99–100
- Sparse matrices (稀疏矩阵), 666
- Sparse small-world graphs (稀疏小世界图), 693
- Sparse vectors (稀疏向量), 666
- Spatial vectors (空间向量), 442–445
- Specification problem (规范问题)
- APIs (应用程序编程接口), 430
  - formal languages (形式语言), 722
  - programs (程序), 596
- Speed (速度)
- clocks (时钟), 1058
  - in performance (性能), 507–508
- Spider traps (蜘蛛陷阱), 176
- Spira mirabilis (等角螺线), 398
- Spiral program (程序 Spiral), 398–399
- Spirographs (螺旋体), 167
- Split program (程序 Split), 358, 360
- Spreadsheets (电子表格), 108
- Sqrt program (程序 Sqrt), 65–67
- Square brackets ([])

- arrays (数组), 91, 106
- regular expressions (正则表达式), 731
- Square roots (平方根)
  - computing (计算), 65–67
  - double value (双值), 25
- Squares, Albers (阿尔伯斯方块), 341–342
- Squaring Markov chains (马尔可夫链自乘), 179–180
- SR flip-flops (SR 触发器), 1050
- ST library (ST 库), 625–627
- Stable circuits with feedback (含有反馈的稳定电路), 1049
- Stack program (程序 Stack), 583–585
- StackOfStrings program (程序 StackOfStrings), 568
- StackOverflowError, 282
- Stacks (栈)
  - arithmetic expression evaluation (算术表达式计算), 586–589
  - arrays (数组), 568–570, 578–581
  - function calls (函数调用), 590–591
  - linked lists (链表), 574–576
  - overview (概述), 566
  - parameterized types (参数化类型), 582–586
  - pushdown (入栈), 567–568
  - stack-based languages (基于栈的语言), 590
  - summary (总结), 608
- Standard audio (标准音频)
  - concert A, 155
  - description (描述), 126, 128–129
  - music example (音乐示例), 157–158
  - notes (注释), 156
  - overview (概述), 155
  - sampling (采样), 156–157
  - saving files (保存文件), 157
  - summary (总结), 159
- Standard deviation (标准差), 246
- Standard drawing (标准绘图)
  - control commands (控制命令), 145–146
  - description (描述), 126, 128–129
  - double buffering (双缓冲区), 151
  - filtering data to (过滤数据), 146–147
  - function graphs (函数图), 148
  - outline and filled shapes (轮廓和填充形状), 149
  - overview (概述), 144–145
  - summary (总结), 159
  - text and color (文本和颜色), 150
- Standard input (标准输入)
  - arbitrary size (任意尺寸), 137–138
  - description (描述), 126, 128–129
  - formatted (格式化), 135
  - interactive (交互式), 135–136
  - machine language (机器语言), 936–938
  - multiple streams (多个流), 143
  - overview (概述), 132–133
  - redirecting (重定向), 140–141
  - summary (总结), 159
  - typing (键入), 134
  - virtual machines (虚拟机), 969–970
- Standard output (标准输出)
  - description (描述), 127
  - formatted (格式化), 130–132
  - machine language (机器语言), 934–936
  - multiple streams (多个流), 143
  - overview (概述), 129–130
  - pipng (管道), 141–143
  - redirecting (重定向), 139–140
  - summary (总结), 159
  - virtual machines (虚拟机), 969–970
- Standard statistics (标准统计)(b), 244–250
- Standards, API (标准 API), 429
- Start codons (起始密码子), 336
- Statements (语句)
  - assignment (赋值), 17
  - blocks (块), 50
  - declaration (声明), 15–16
  - methods (方法), 5
- States (状态)
  - DFAs (确定有限状态自动机), 738–739
  - NFAs (非确定有限状态自动机), 744–746
  - objects (对象), 340
  - Turing machines (图灵机), 766–772
  - virtual machines (虚拟机), 968
- Static methods (静态方法), 191–192
  - accessing (访问), 227–229
  - arguments (参数), 197
  - for code organization (代码组织), 205–206
  - control flow (控制流), 193–195
  - defining (定义), 193, 196

- function-call traces (函数调用跟踪), 195
- function calls (函数调用), 197
- implementation examples (实现示例), 199
- vs. instance (对比实例), 340
- libraries (库), 198
- passing arguments (传递参数), 207–210
- returning values (返回值), 207–210
- Side effects (副作用), 201
  - summary (总结), 215
  - superposition example (叠加例子), 211–215
  - terminology (术语), 195–196
  - variable scope (变量作用域), 200
- Static variables (静态变量), 284
- Statistical polling (统计调查), 167
- Statistics (统计), 244–250
- StdArrayIO library (StdArrayIO 库), 237–238
- StdAudio library (StdAudio 库), 128–129, 155
- StdDraw library (StdDraw 库), 128–129, 144–145, 150, 154
- StdIn library (StdIn 库), 128–129, 132–133
- StdOut library (StdOut 库), 129–131
- StdRandom program (程序 StdRandom), 232–236
- StdStats program (程序 StdStats), 244–247
- StockAccount program (程序 StockAccount), 410–413
- StockQuote program (程序 StockQuote), 358–359
- Stop codons (终止密码子), 336
- Stopwatch program (程序 Stopwatch), 390–391
- Store instruction (存储指令), 938, 1080
- Stored-program computers (存储程序计算机), 922–924
- Streams (流)
  - input (输入), 354–355
  - output (输出), 355
  - screen scraping (屏幕抓取), 357–359
- Stress testing (压力测试), 236
- Strings and String data type (字符串与字符串数据类型)
  - alphabet symbols (字母符号), 718
  - API (应用程序编程接口), 332–333
  - binary (二进制), 718–719
  - circular shifts (循环移位), 375
  - concatenation (拼接), 19–20, 723–724
  - conversion codes (转换代码), 131–132
  - conversions (转换), 21, 453
  - description (描述), 14–15
  - genomics application (基因组学应用), 336–340
  - immutable types (不可变类型), 439–440
  - input (输入), 133
  - internal storage (内部存储), 37
  - invoking instance methods (调用实例方法), 334
  - memory (内存), 515
  - objects (对象), 333–334
  - overview (概述), 331
  - parsing (解析), 880–882
  - prefix-free (无前缀), 564
  - representation (表示), 882–883
  - as sequence of characters (字符串序列), 19
  - shortcuts (捷径), 334–335
  - string replacement systems (字符串替换系统), 795
  - unions (并集), 723
  - variables (变量), 333
  - vertices (顶点), 675
  - working with (共同作用), 19–21
- Strogatz, Stephen (史蒂芬·斯特罗格茨), 670, 693, 713
- Structured programming (结构化编程), 926
- Stub methods (存根方法), 303
- Subclassing inheritance (子类继承), 452–457
- Subgraphs, induced (诱导子图), 705
- Subset sum problem (子集和问题)
- intractability (难解性), 827–828
  - NP, 834, 838
- Subtraction (减法)
  - floating-point numbers (浮点数), 24–26
  - integers (整数), 22
  - negative numbers (负数), 887
- Subtrees (子树), 640, 651
- Subtyping inheritance (子类型继承), 446–451
- Sum-of-powers conjecture (幂和猜想), 89
- Sum-of-products (积之和)
  - adders (加法器), 1028
  - boolean representation (布尔表示), 996–997
  - circuits (电路), 1024–1027
- Sums, finite (有限和), 64–65
- Superclasses (超类), 452
- Superposition (叠加)

- force vectors (力向量), 483
- sound waves (声波), 211–215
- Swirl filters (漩涡过滤器), 379
- Switch control lines (开关控制线), 1005
- Switch statements (开关语句), 74–75
- Switches (切换)
  - bus muxes (总线多路复用器), 1036
  - circuit models (电路模型), 1002, 1005–1006
  - demultiplexers (多路分配器), 1022
  - gates (门), 1013
  - multiplexers (多路复用器), 1020
  - TOY machine (TOY 机), 916–917
- Switching circuit analysis (开关电路分析), 1007
- Switching time of gates (门的切换时间), 1013
- Symbol tables (符号表)
  - APIs (应用程序编程接口), 625–627
  - BSTs (二叉搜索树)
    - dictionary lookup (字典查找), 628–632
    - graphs (图), 676
    - hash tables (散列表), 636–639
    - implementations (实现), 635–636
    - indexing (索引), 632–634
    - machine language (机器语言), 944
    - overview (概述), 624–625
    - perspective (展望), 654
    - sets (集合), 652–653
- Symbolic names in assembly (汇编语言的符号名称), 981
- Symbols (符号)
  - definition (定义), 757
  - description (描述), 718–719
  - DFA (确定有限状态自动机), 738
  - NFA (非确定有限状态自动机), 744
  - regular expressions (正则表达式), 724
  - Turing machines (图灵机), 766–767
- Symmetric order in BSTs (BST 中的对称顺序), 640
- Symmetric property (对称属性), 454
- Syntax errors (语法错误), 10–11
- hash (散列), 636–639
- symbol (符号)
- Tabs (制表符)
  - compiler considerations (编译考虑), 10
  - escape sequences (转义序列), 19
- Tape and tape readers (纸带和纸带阅读器)
  - DFA (确定有限状态自动机), 738–739
  - Turing machines (图灵机), 766–769, 774–776
- Tape program (程序 Tape), 776
- Taylor series approximations (泰勒级数近似值), 204
- Templates (模板), 50
- TenHellos program (程序 TenHellos), 54–55, 60
- Terminal windows (终端窗口), 127
- Terms, glossary for (术语表), 1097–1101
- Terrain analysis (地形分析), 167
- Testing (测试)
  - for bugs (错误), 318
  - importance (重要性), 701
  - percolation case study (渗透案例研究), 305–308
- Text (文本)
  - drawings (绘图), 150
  - printing (打印), 5, 10
- Text editors (文本编辑器), 3
- Theory of computing (计算理论), 715–717
- this keyword (this 关键字), 445
- Thompson, Ken (肯·汤普森), 735
- $3n+1$  problem ( $3n+1$  问题), 296–297
- ThreeSum program (程序 ThreeSum), 497–502
- Throwing exceptions (抛出异常), 465–466
- Thue word problem (图厄词问题), 819
- Ticks, clock (时钟), 1058
- Tilde notation (波浪线表示法), 500
- Tildes (~)
  - bitwise operations (按位运算), 891
  - boolean type (布尔类型), 991
  - frequency analysis (频率分析), 500
- Time (时间)
  - exponential (指数), 272–273, 823
  - performance (性能)
  - polynomial (多项式), 823
  - Stopwatch timers (秒表计时器), 390–391
- TimePrimitives program (程序 TimePrimitives), 519
- Timesharing (分时操作), 965

## T

## Tables (表)

- of functions (函数), 907–908

- Tools, building (构建工具), 320
- Top-level domains (顶级域名), 375
- toString() method (方法 toString())
  - Charge, 383, 387
  - Color, 343
  - Complex, 403, 405
  - Convert, 881–882
  - Counter, 436–437
  - description (描述), 339
  - Graph, 678–679
  - linked lists (链表), 574, 577
  - Object, 453
  - Sketch, 459
  - Tape, 776
  - Vector, 443
- Total orderings (全序), 546
- Totality problem (总体问题), 811–812
- Towers of Hanoi problem (汉诺塔问题), 268–272
- TOY machine (TOY 机)
  - arithmetic logic unit (算术逻辑单元), 910
  - conditionals and loops (条件和循环), 918–921
  - family of computers (计算机系列), 972–977
  - fetch–increment–execute cycle (取指–增量–执行周期), 910–911
  - first program (第一个程序), 914–915
  - historical note (历史记录), 907–908
  - instruction register (指令寄存器), 910
  - instructions (指令), 909, 911–913
  - in Java (Java 中), 966–972
  - machine-language programming (机器语言编程)
  - memory (内存), 908–909
  - operating (操作), 916–917
  - overview (概述), 906–907
  - program counter (程序计数器), 910
  - registers (寄存器), 909
  - stored-program computer (存储程序计算机), 922–924
  - virtual (虚拟)
  - von Neumann machines (冯·诺依曼计算机), 924–925
- TOY program (程序 TOY), 967
- TOY-8 machine (TOY-8 计算机), 974–975
  - basic parameters (基本参数), 1070
  - control circuit (控制电路), 1080–1082
  - CPU (中央处理器), 1076–1080
  - instruction set (指令集), 1070–1071
  - perspective (展望), 1084–1087
  - sum.toy program (程序 sum.toy), 1071–1072, 1082–1083
- TOY-64 machine (TOY-64 计算机), 973–974
- Tracing (追踪)
  - function-call (函数调用), 195
  - programs with random() (程序 random()), 103
  - variable values (变量值), 18, 56–57
- Transfer of control (转换控制), 193–195
- Transistors (晶体管), 1006
- Transition matrices (转换矩阵), 172–173
- Transition program (程序 Transition), 172–173
- Transitions (转移)
  - DFA (确定有限状态自动机), 738–739
  - NFA (非确定有限状态自动机), 744–746
  - Turing machines (图灵机), 766–767
- Transitive property (传递性)
  - comparisons (比较), 546
  - equivalence (等价), 454
  - polynomial-time reduction (多项式时间归约), 843
- Transposition of arrays (数组的转置), 120
- Traveling salesperson problem (旅行商问题), 862
- Traversal (遍历)
  - binary search trees (二叉搜索树), 649–650
  - linked lists (链表), 574, 577
- TreeMap library (TreeMap 库), 655
- Tree nodes (树节点), 269
- Trees (树)
  - BSTs (二叉搜索树)
  - function-call (函数调用), 269, 271
  - H-trees (H 树), 276–277
  - shortest paths (最短路径), 688–689
- Triangles (三角形)
  - drawing (绘图), 144–145
  - right (直角), 199
  - Sierpinski (谢尔宾斯基), 239–240
- Trigonometric functions (三角函数), 256
- Truth tables (真值表), 26–27, 988–989
- TSP problem (TSP 问题), 862
- Turing, Alan (艾伦·图灵), 766
  - bio (简历), 410–411, 717

- code breaking (代码破坏), 907
  - von Neumann influenced by (冯·诺依曼的影响者), 924–925
  - Turing-complete models (图灵完全模型), 794
  - Turing machines (图灵机)
    - binary adders (二进制加法器), 771
    - binary incrementers (二进制递增器), 769–771
    - compact trace format (紧凑的跟踪格式), 770
    - constant factor (常数因子), 824
    - efficiency (效率), 772
    - frequency count (频率计数器), 772–773
    - model (模型), 766–769
    - overview (概述), 766
    - related machines (关联机器), 770–771
    - restrictions (限制), 792–793
    - SAT problem (SAT 问题), 836
    - universal (通用), 789–790
    - universal virtual (通用虚拟), 774–779
    - universal virtual DFAs (通用虚拟 DFA), 789
    - universality (普遍性)
    - variations (变量), 791–794
  - TuringMachine program (程序 TuringMachine), 777–778
  - Turtle program (程序 Turtle), 394–396
  - Twenty questions game (20 个问题游戏), 135–136, 533–535
  - TwentyQuestions program(程序 TwentyQuestions), 135–136
  - Two-dimensional arrays (二维数组)
    - description (描述), 90
    - initialization (初始化), 106–107
    - matrices (矩阵), 109–110
    - memory (内存), 107, 516
    - output (输出), 107
    - overview (概述), 106
    - ragged (交错), 111
    - self-avoiding walks (自回避行走), 112–115
    - setting values (设定值), 108
    - spreadsheets (电子表格), 108
  - Two's complement (二进制补码), 38, 886–888
  - Type arguments (类型参数), 585, 611
  - Type conversions (类型转换), 34–35
  - Type parameters (类型参数), 585
  - Type safety (类型安全), 18
  - Types (类型)
- ## U
- Unboxing (拆箱), 457, 585–586
  - Undirected graphs (无向图), 675
  - Unicode characters (Unicode 字符)
    - description (描述), 19
    - overview (概述), 894–895
    - strings (字符串), 37
  - Uniform random numbers (统一随机数), 199
  - Uninitialized variables (未初始化的变量), 94, 339
  - Union operation in REs (正则表达式中的交集操作), 723
  - Unit testing (单元测试), 235
  - Universal models (通用模型), 794–797
  - Universal sets (通用集合)
    - elementary functions (基本功能), 1001
    - gates (门), 1045
  - Universal Turing machines (UTMs)(通用图灵机), 789–790
  - Universal virtual DFAs (通用虚拟 DFA), 741–743
  - Universal virtual TMs (通用虚拟 TM), 774–779
  - Universality (普遍性)
    - algorithms (算法), 786–787
    - Church–Turing thesis (邱奇–图灵论题), 790–791
    - overview (概述), 786
    - programs processing programs (程序处理程序), 788–790
    - Turing machine variations (图灵机变量), 791–794
    - universal models (通用模型), 794–797
    - virtual DFA/NFA (虚拟 DFA/NFA), 788–789
  - Universe program (程序 Universe), 483–487
  - Unreachable code error (无法访问的代码错误), 216
  - Unsigned integers (无符号整数), 884
  - Unsolvability proof (无法证明), 810
  - Unsolvable problems (无解问题), 430
    - blank tape halting problem(空白纸带停机问题), 820
    - definite integration (定积分), 816
    - description (描述), 806
    - examples (示例), 815
    - halting problem (停机问题), 808–810



Hilbert's 10th problem (希尔伯特十大问题), 816  
 implications (影响), 816–817  
 liar's paradox (说谎者悖论), 807–808  
 optimal data compression (最佳数据压缩), 814  
 optimizing compilers (优化编译器), 814  
 Post correspondence (邮寄通信), 813–814  
 program equivalence (程序等价性), 812  
 totality (总和), 811–812  
 Upper bounds (上限), 825  
 Upscaling in image processing (图像处理中的升级), 349  
 UseArgument program (程序 UseArgumen), 7–8  
 User-defined libraries (用户自定义模块), 230  
 UTF-8 encoding (UTF-8 编码), 895  
 UTMs, 789–790

## V

Validate program (程序 Validate), 729  
 Validity checking (有效性检查), 732  
 Values (值)  
   array (数组), 95–96  
   data types (数据类型), 14, 331  
   passing arguments by (传递参数), 207, 210, 364–365  
   precomputed (预先计算), 99–100  
   symbol tables (符号表), 624–626  
 Variables (变量)  
   assignment statements (赋值语句), 17  
   boolean (布尔), 987, 994–997  
   compound assignments (组合赋值), 60  
   constants (常数), 16  
   description (描述), 15–16  
   initial values (初始值), 415  
   inline initialization (内联初始化), 18  
   instance (实例), 384  
   within methods (在方法中), 196, 386–388  
   names (名称), 16  
   scope (作用域), 60, 200  
   shadow (影子), 419  
   static (静态), 284  
   string (字符串), 333  
   tracing values (追踪值), 18  
   uninitialized (未初始化), 339

Vector images (向量图形), 346  
 Vector program (程序 Vector), 443–445, 515  
 Vectors (向量)  
   arrays (数组), 92  
   cross products (向量积), 472  
   dot products (点积), 92, 442–443  
   matrix–vector multiplication (矩阵向量乘法), 110  
   n-body simulation (多体模拟), 479–480  
   sparse (稀疏), 666  
   spatial (空间的), 442–445  
   vector–matrix multiplication (向量–矩阵乘法), 110, 180  
 Vertex cover problem (顶点覆盖问题)  
   intractability (难解性), 828  
   NP-completeness (NP 完全性), 846–847  
   NP search problems (NP 搜索问题), 834, 842  
 Vertical bars (|)  
   bitwise operations (按位操作), 891–892  
   boolean type (布尔类型), 26–27, 991  
   piping (管道), 141  
   regular expressions (正则表达式), 724  
 Vertical OR gates (垂直或门), 1023  
 Vertical percolation (垂直渗透原理), 305–306  
 Vertices (顶点)  
   bipartite graphs (二分图), 682  
   creating (创建), 676  
   eccentricity (偏心距), 711  
   graphs (图), 671, 674  
   isolated (孤立的), 703  
   names (命名), 675  
   PathFinder, 683  
   String, 675  
 Virtual machines (虚拟机)  
   booting (引导), 959–960, 968–969  
   cautions (警告), 961–963  
   and cloud computing (云计算), 924  
   description (描述), 965  
   dumping (释放), 960–961  
   instructions (指令), 966–967  
   Moore's law (摩尔定律), 971  
   overview (概述), 958–959  
   program development (程序发展), 970–971

- programs that process programs (程序处理程序), 964–966
  - running (运行), 969
  - standard input (标准输入), 969–970
  - standard output (标准输出), 969–970
  - states (状态), 968
  - TOY machine family (TOY 计算机家族), 972–977
  - universal virtual DFAs (通用虚拟 DFA), 742
  - universal virtual TM (通用虚拟 TM), 774–779
  - Viruses (病毒), 963
  - Viterbi algorithm (维特比算法), 286
  - void keyword (void 关键字), 201, 216
  - Volatility (波动)
    - Black-Scholes formula (布莱克-斯科尔斯期权计价公式), 565
    - Brownian bridges (布朗桥), 278, 280
  - Von Neumann, John (约翰·冯·诺依曼), 906
    - ballistics tables (弹道表), 907
    - ENIAC improvements (ENIAC 改进), 924–925
    - Gödel letter (哥德尔信件), 840
    - mergesort (归并排序), 554
  - Von Neumann architecture (冯·诺依曼结构), 790, 906, 924–925
  - Voting machine errors (投票机错误), 436
- W**
- Walks (行走)
    - random (随机), 112–115, 710
  - Watson-Crick palindrome (Watson-Crick 互补碱基回文), 374
  - Watts, Duncan (邓肯·瓦茨), 670, 693, 713
  - Watts-Strogatz graph model (Watts-Strogatz 图模型), 713
  - .wav format (.wav 格式), 157
  - Wave filters (滤波器), 379
  - Web graphs (Web 图), 695
  - Web pages (网页), 170
    - indexes searches (索引查找), 634
    - preferential attachment (首选链接), 713
  - Weighing objects (物体称重法), 540–541
  - Weighted averages (加权平均值), 120
  - Weighted superposition (加权叠加), 212
  - while loops (while 循环), 53–59
    - examples (示例), 61
    - nesting (嵌套), 62
  - Whitelists, binary searches for (白名单二分查找法), 540
  - Whitespace characters (空白字符)
    - compiler considerations (编译器考虑), 10
    - input (输入), 135
  - Wide interfaces (宽接口)
    - APIs (应用程序编程接口), 430
    - examples (示例), 610–611
  - Wildcard operation in Res (RE 中的通配符操作), 724
  - Wiles, Andrew (安德鲁·威尔斯), 722
  - Wind chill (风寒指数), 47
  - Wires (电线)
    - circuit models (电路模型), 1002–1004
    - gates (门), 1013
  - Word ladders (词梯), 710
  - Words (单词)
    - binary representation (二进制表示), 875
    - computer (计算机), 874
    - memory size (内存大小), 513
    - size (规模大小), 897
  - Worst-case performance (最坏情况下的性能)
    - big-O notation (大 O 符号), 520–521
    - binary search trees (二叉搜索树), 648
    - description (描述), 512
    - insertion sort (插入排序), 544
    - intractability (难解性), 825
    - NP-completeness (NP 完全性), 852
  - Wrapper types (包装类型)
    - autoboxing (自动装箱), 585–586
    - references (引用), 369, 457
  - Write control lines (写控制线)
    - CPU (中央处理器), 1079–1080
    - memory bits (内存位), 1056
    - register bits (寄存器位), 1051

## X

- XOR circuits (异或电路)
  - in arithmetic logic units (算术逻辑单元), 1031
  - sum-of-products (积之和), 1024–1025
- xor (exclusive or) operation (异或操作), 891–892, 913

## Y

- Y2K problem (千年问题), 435, 976
- Young tableaux (杨氏矩阵), 530

## Z

- Zero-based indexing (零基 (从 0 开始) 索引), 92
- Zero crossings (零交叉点), 164
- Zero extension convention (零扩展惯例), 899
- 0/1 ILP problem (0/1ILP 问题), 831, 835
  - NP-completeness (NP 完全性), 845–846
  - vertex cover problem (顶点覆盖问题), 842
- Zeros, leading (前面的 0), 883
- ZIP codes (邮政编码), 435
- Zipf's law (齐普夫定律), 556

# API

**public class Math**

---

|                                       |             |
|---------------------------------------|-------------|
| <b>double abs(double a)</b>           | a 的绝对值      |
| <b>double max(double a, double b)</b> | a 和 b 中的较大者 |
| <b>double min(double a, double b)</b> | a 和 b 中的较小者 |

注1: int、long和float类型数据也可以使用abs、max()和min()方法

|                                 |           |
|---------------------------------|-----------|
| <b>double sin(double theta)</b> | theta的正弦值 |
| <b>double cos(double theta)</b> | theta的余弦值 |
| <b>double tan(double theta)</b> | theta的正切值 |

注2: 角度以弧度表示 使用toDegrees()和toRadians()进行弧度角度的转换

注3: 使用asin()、acos()和atan()计算反三角函数。

|                                       |                                |
|---------------------------------------|--------------------------------|
| <b>double exp(double a)</b>           | 指数操作 ( $e^a$ )                 |
| <b>double log(double a)</b>           | 自然对数 ( $\log_e a$ 或者 $\ln a$ ) |
| <b>double pow(double a, double b)</b> | 计算a的b次幂 ( $a^b$ )              |
| <b>long round(double a)</b>           | 将a向下取整                         |
| <b>double random()</b>                | [0, 1)范围内的一个随机值                |
| <b>double sqrt(double a)</b>          | a的平方根                          |
| <b>double E</b>                       | 欧拉数e的值 (常量)                    |
| <b>double PI</b>                      | $\pi$ 的值 (常量)                  |

---

**public class String**


---

|                                                    |                            |
|----------------------------------------------------|----------------------------|
| <code>String(String s)</code>                      | 创建一个与s值相同的字符串              |
| <code>int length()</code>                          | 字符的个数                      |
| <code>char charAt(int i)</code>                    | 索引下标为i的字符                  |
| <code>String substring(int i, int j)</code>        | 索引下标从i到j-1之间的字符串           |
| <code>boolean contains(String sub)</code>          | 此字符串是否包含sub string?        |
| <code>boolean startsWith(String pre)</code>        | 此字符串是否以pre开头?              |
| <code>boolean endsWith(String post)</code>         | 此字符串是否以post结尾?             |
| <code>int indexOf(String p)</code>                 | 第一次出现p的位置索引                |
| <code>int indexOf(String p, int i)</code>          | 在索引位置i之后第一次出现的pattern的索引下标 |
| <code>String concat(String t)</code>               | 此字符串后添加t                   |
| <code>int compareTo(String t)</code>               | 字符串比较                      |
| <code>String replaceAll(String a, String b)</code> | 此字符串中的a用b来替换               |
| <code>String[] split(String delim)</code>          | 字符串被delim分割后的子字符串组         |
| <code>boolean equals(String t)</code>              | 此字符串的值是否与t的值相同             |

---

**public class System.out/StdOut/Out**


---

|                                               |                  |
|-----------------------------------------------|------------------|
| <code>Out(String name)</code>                 | 从name创建输出流       |
| <code>void print(String s)</code>             | 打印s              |
| <code>void println(String s)</code>           | 打印s并另起一行         |
| <code>void println()</code>                   | 打印新行             |
| <code>void printf(String format, ... )</code> | 根据格式规范将参数打印到标准输出 |

注：对于System.out/Stdout，方法是静态的，且构造函数未使用。

```
public class StdIn/In
```

| <b>In(String name)</b>             | 从name创建输入流                 |
|------------------------------------|----------------------------|
| 从标准输入读取单个数据的方法                     |                            |
| <b>boolean isEmpty()</b>           | 标准输入是否为空（或仅有空白字符）？         |
| <b>int readInt()</b>               | 读取一个数据，将其转换为int类型，然后返回     |
| <b>double readDouble()</b>         | 读取一个数据，将其转换为double类型，然后返回  |
| <b>boolean readBoolean()</b>       | 读取一个数据，将其转换为boolean类型，然后返回 |
| <b>String readString()</b>         | 读取数据并将其作为字符串返回             |
| 从标准输入读取字符的方法                       |                            |
| <b>boolean hasNextChar()</b>       | 标准输入中是否还有字符尚未读取？           |
| <b>char readChar()</b>             | 从标准输入读取一个字符并返回             |
| 从标准输入读取行的方法                        |                            |
| <b>boolean hasNextLine()</b>       | 标准输入中是否有下一行尚未读取？           |
| <b>String readLine()</b>           | 读取行的其余部分并将其作为字符串返回         |
| 读取其余标准输入的方法                        |                            |
| <b>int[] readAllInts()</b>         | 读取所有剩余的数据并将其作为int数组返回      |
| <b>double[] readAllDoubles()</b>   | 读取所有剩余的数据并将其作为double数组返回   |
| <b>boolean[] readAllBooleans()</b> | 读取所有剩余的数据并将其作为boolean数组返回  |
| <b>String[] readAllStrings()</b>   | 读取所有剩余的数据并将其作为String数组返回   |
| <b>String[] readAllLines()</b>     | 读取所有剩余的行并将其作为String数组返回    |
| <b>String readAll()</b>            | 读取其余的输入并将它作为一个字符串返回        |

注1：对于StdIn，方法是静态的，且构造函数未使用。

注2：数据是非空白字符的最大序列。

注3：在读取数据之前，任何前导空格都将被丢弃。

注4：有类似的方法来读取byte、short、long和float类型的值。

注5：读取输入的每个方法如果无法读取下一个值，将抛出运行时异常，可能因为没有更多的输入或因为输入与预期类型不匹配。

```
public class StdDraw
```

```
 Draw()
```

创建一个新的Draw对象

绘图命令

```
void line(double x0, double y0, double x1, double y1)
void point(double x, double y)
void circle(double x, double y, double radius)
void filledCircle(double x, double y, double radius)
void square(double x, double y, double radius)
void filledSquare(double x, double y, double radius)
void rectangle(double x, double y, double r1, double r2)
void filledRectangle(double x, double y, double r1, double r2)
void polygon(double[] x, double[] y)
void filledPolygon(double[] x, double[] y)
void text(double x, double y, String s)
```

控制命令

```
void setXscale(double x0, double x1) 将x坐标重置为 (x0, x1)
void setYscale(double y0, double y1) 将y坐标重置为 (y0, y1)
void setPenRadius(double radius) 将笔半径设置为radius
void setPenColor(Color color) 设置笔的颜色为color
void setFont(Font font) 设置文本字体为font
void setCanvasSize(int w, int h) 将画布大小设置为宽w, 高h
void enableDoubleBuffering() 启用双缓冲
void disableDoubleBuffering() 关闭双缓冲

void show() 将缓冲区画布复制到屏幕画布

void clear(Color color) 清空画布并将画布颜色设置为color
void pause(int dt) 暂停dt毫秒
void save(String filename) 保存为.jpg文件或.png文件
```

注1: 对于StdDraw, 方法是静态的, 且构造函数未使用。

注2: 具有相同名称但无参数的方法重置为默认值。



**public class StdAudio**

---

|                                               |                       |
|-----------------------------------------------|-----------------------|
| <b>void play(String filename)</b>             | 播放给定的.wav文件           |
| <b>void play(double[] a)</b>                  | 播放给定的声波               |
| <b>void play(double x)</b>                    | 将x作为采样样本播放1 / 44 100秒 |
| <b>void save(String filename, double[] a)</b> | 保存到一个.wav文件           |
| <b>double[] read(String filename)</b>         | 从一个.wav文件读取           |

**public class Stopwatch**

---

|                             |                      |
|-----------------------------|----------------------|
| <b>Stopwatch()</b>          | 创建一个新的秒表对象并运行        |
| <b>double elapsedTime()</b> | 自秒表创建以来所经过的时间（以秒为单位） |

**public class Picture**

---

|                                            |                     |
|--------------------------------------------|---------------------|
| <b>Picture(String filename)</b>            | 从文件创建一个图片           |
| <b>Picture(int w, int h)</b>               | 创建一个w×h的空白图片        |
| <b>int width()</b>                         | 返回图片的宽度             |
| <b>int height()</b>                        | 返回图片的高度             |
| <b>Color get(int col, int row)</b>         | 返回像素（col, row）的颜色   |
| <b>void set(int col, int row, Color c)</b> | 给c设置像素（col, row）的颜色 |
| <b>void show()</b>                         | 在窗口上展示图片            |
| <b>void save(String filename)</b>          | 将图片保存为文件            |

**public class StdRandom**


---

|                                                 |                            |
|-------------------------------------------------|----------------------------|
| <b>void setSeed(long seed)</b>                  | 设置种子以获得可重现的结果              |
| <b>int uniform(int n)</b>                       | 0和 $n-1$ 之间的整数             |
| <b>double uniform(double lo, double hi)</b>     | lo和hi之间的浮点数                |
| <b>boolean bernoulli(double p)</b>              | true的概率为p, false的概率为 $1-p$ |
| <b>double gaussian()</b>                        | 高斯分布, 均值0, 标准差1            |
| <b>double gaussian(double mu, double sigma)</b> | 高斯分布, 均值mu, 标准差sigma       |
| <b>int discrete(double[] p)</b>                 | 以 $p[i]$ 的概率生成i            |
| <b>void shuffle(double[] a)</b>                 | 随机混排数组a[]中的元素              |

**public class StdArrayIO**


---

|                                  |           |
|----------------------------------|-----------|
| <b>double[] readDouble1D()</b>   | 读取一维浮点数数组 |
| <b>double[][] readDouble2D()</b> | 读取二维浮点数数组 |
| <b>void print(double[] a)</b>    | 输出一维浮点数数组 |
| <b>void print(double[][] a)</b>  | 输出二维浮点数数组 |

注1: 以1D结尾的函数, 其格式是1个整数 $n$ 后跟 $n$ 个值。

注2: 以2D结尾的函数, 其格式是2个整数 $m$ 和 $n$ , 后面跟着 $m \times n$ 个值, 以行排序

注3: API中还包括基于int和boolean的相同方法。

**public class StdStats**


---

|                                    |                                                      |
|------------------------------------|------------------------------------------------------|
| <b>double max(double[] a)</b>      | 最大值                                                  |
| <b>double min(double[] a)</b>      | 最小值                                                  |
| <b>double mean(double[] a)</b>     | 平均值                                                  |
| <b>double var(double[] a)</b>      | 样本方差                                                 |
| <b>double stddev(double[] a)</b>   | 样本标准差                                                |
| <b>double median(double[] a)</b>   | 中位数                                                  |
| <b>void plotPoints(double[] a)</b> | 在 $(i, a[i])$ 处绘制点                                   |
| <b>void plotLines(double[] a)</b>  | 绘制连接点 $(i, a[i])$ 的线段                                |
| <b>void plotBars(double[] a)</b>   | <i>plot bars to points at <math>(i, a[i])</math></i> |

注: 也包含其他数值类型的重载实现

---

```
public class Stack<Item> implements Iterable<Item>
```

---

|                                   |                |
|-----------------------------------|----------------|
| <code>Stack()</code>              | 创建一个空栈         |
| <code>boolean isEmpty()</code>    | 栈是否为空          |
| <code>int size()</code>           | 栈中数据项项数        |
| <code>void push(Item item)</code> | 插入一个数据项到栈中     |
| <code>Item pop()</code>           | 将最近插入的数据项删除并返回 |

---

```
public class Queue<Item> implements Iterable<Item>
```

---

|                                      |                |
|--------------------------------------|----------------|
| <code>Queue()</code>                 | 创建一个空队列        |
| <code>boolean isEmpty()</code>       | 队列是空的吗?        |
| <code>int size()</code>              | 将一个项插入队列中      |
| <code>void enqueue(Item item)</code> | 移除并返回队列中最早加入的项 |
| <code>Item dequeue()</code>          | 队列中项的个数        |

---

```
public class SET<Key extends Comparable<Key>> implements Iterable<Key>
```

---

|                                        |           |
|----------------------------------------|-----------|
| <code>SET()</code>                     | 新建一个空的集合  |
| <code>boolean isEmpty()</code>         | 集合是否为空?   |
| <code>int size()</code>                | 集合中键的数量   |
| <code>void add(Key key)</code>         | 向集合中添加键   |
| <code>void remove(Key key)</code>      | 从集合中删除键   |
| <code>boolean contains(Key key)</code> | 该键是否在集合中? |

# ST

```
public class ST<Key extends Comparable<Key>, Value>
```

---

|                              |                    |
|------------------------------|--------------------|
| ST()                         | 创建一个空的符号表          |
| void put(Key key, Value val) | 将val与key建立关联       |
| Value get(Key key)           | 获取与key相关联的值        |
| void remove(Key key)         | 删除键key及其对应的值       |
| boolean contains(Key key)    | 是否存在对应于key的值       |
| int size()                   | 键值对的数量             |
| Iterable<Key> keys()         | 按照排好序的顺序返回符号表中所有的键 |
| Key min()                    | 返回最小的键             |
| Key max()                    | 返回最大的键             |
| int rank(Key key)            | 小于key的键的数量         |
| Key select(int k)            | 从小到大排序的第k个键        |
| Key floor(Key key)           | 小于等于key的最大键        |
| Key ceiling(Key key)         | 大于等于key的最小键        |

```
public class Graph
```

---

|                                       |            |
|---------------------------------------|------------|
| Graph()                               | 创建一个空图     |
| Graph(In in, String delimiter)        | 从输入流中读取图   |
| void addEdge(String v, String w)      | 增加v-w边     |
| int V()                               | 顶点数        |
| int E()                               | 边数         |
| Iterable<String> vertices()           | 图中的顶点      |
| Iterable<String> adjacentTo(String v) | v的邻接顶点     |
| int degree(String v)                  | v的邻接顶点的数量  |
| boolean hasVertex(String v)           | v是图中的顶点吗?  |
| boolean hasEdge(String v, String w)   | v-w是图中的边吗? |

## 推荐阅读



### 数据结构与抽象：Java语言描述（原书第5版）

作者：Frank M.Carrano 等 ISBN：978-7-111-63637-3 定价：139.00元

### 数据结构与算法经典问题解析（原书第2版）

作者：Narasimha Karumanchi ISBN：978-7-111-61241-4 定价：79.00元

### Java语言程序设计与数据结构（进阶篇）（原书第11版）

作者：Y. Daniel Liang

ISBN：978-7-111-61241-4 定价：99.00元



# 计算机科学导论 跨学科方法

Computer Science An Interdisciplinary Approach

本书基于“编程是21世纪科学和工程领域教育的重要组成部分”这一趋势，以Java语言为工具，通过跨学科方式对计算机科学进行了生动有趣的介绍，非常适合作为学生和对计算机感兴趣的读者的入门教材。读者只需具备科学和数学的基础知识，便可通过本书开启计算机科学之旅！

## 本书特点

- 本书以解决实际问题为宗旨，关注如何通过计算机和编程来解决数学、科学、工程等领域的实际问题。同时，帮助读者揭开计算的神秘面纱，建立对计算机科学领域的系统认知。
- 本书第一部分基于Java语言，围绕编程的三个阶段组织内容，使读者了解编程的基本元素、函数和面向对象编程的基本知识。第二部分主要介绍算法和数据结构、计算理论和计算机体系结构等高级主题。此外，本书还强调了计算机发展的历史知识和计算思想的发展与应用。
- 本书的另一独特之处是通过构建一台“玩具型”计算机，分析程序的执行过程，将计算机底层的工作原理和设计方法巧妙融合，更有效地培养读者对计算机系统的理解与认识。

## 作者简介

**罗伯特·塞奇威克 (Robert Sedgewick)** 普林斯顿大学计算机科学系教授，从1985年开始担任系主任。他于斯坦福大学获得博士学位，师从计算机科学界泰斗高德纳教授。他曾任Adobe公司的董事，并曾在Xerox PARC、IDA和INRIA公司担任研究员。他所编写的算法系列书籍令大批计算机学习者受益匪浅。



**凯文·韦恩 (Kevin Wayne)** 自1998年开始在普林斯顿大学计算机科学系任教，曾荣获2014年ACM杰出教育奖以及普林斯顿大学多个教学方面的奖项。他和Sedgewick教授一起开设计算机导论和数据结构与算法的入门课程，他们共同开设的MOOC课程已吸引了超过100万的学习者。



www.pearson.com



华章教育服务微信号



投稿热线: (010) 88379604  
读者信箱: hzsj@hzbook.com  
客服电话: (010) 88361066 88379833 68326294

华章网站: www.hzbook.com  
网上购书: www.china-pub.com  
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/基础

ISBN 978-7-111-64141-4



9 787111 641414 >

定价: 139.00元